

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

中国Solr领域的资深专家和布道师撰写，权威性毋庸置疑。

以实战为导向，全面、系统、细致、深入地讲解Solr的基础知识、核心技术、进阶知识和扩展知识。



兰小伟 著

The Definitive Guide of Solr

Solr权威指南

下卷



机械工业出版社
China Machine Press

内 容 简 介

本书作者是国内最早接触Solr的技术专家之一，多年一直在Solr的研究、实践和布道的路上不遗余力、乐此不疲。本书立足全球视野，综合Solr技术的最新发展和应用、从业人员的学习曲线，以及中英文资料的供给情况，给自己设定了一个极高的目标：力争在内容的全面性、系统性、深浅度和实战性上超越所有的同类书。从完成的结果上来看，我们的目标接近完成，Solr的基础知识、核心技术、进阶知识和扩展知识悉数包括在内。

全书一共16章，分为上下两卷：

上卷（第1~10章）

全面、系统地讲解了Solr的基础知识和核心技术。包括部署、配置、Solr Core、Solr DIH、全量导入、增量导入、索引、中文分词、查询组件、Solr Facet、高亮、查询建议，以及企业如何在真实的项目中使用Solr。不仅讲解了基本概念和使用方法，而且还分析了各组件的内部工作机制。

下卷（第11~16章）

细致、深入地讲解了Solr的高级知识和拓展知识。

高级知识部分包括：Solr的高级查询及其各种查询技巧，如函数查询、地理空间查询、Facet嵌套等；SolrJ、SolrCloud、Spring Data Solr的使用详解和工作原理；Solr的多种性能优化技巧，如索引的性能优化、缓存的性能优化、查询的性能优化、JVM和Web容器的优化，以及操作系统级别的优化。

拓展知识中首先讲解了Solr的一些比较生僻的知识点，如伪域、多语种索引支持、安全认证，以及Solr 6.x中的SQL接口和Streaming表达式等；然后讲解了Solr与MapReduce、HDFS、Hbase、Kafka、Flume、Storm、Spark等大数据技术的结合使用的集成方法。

實戰



The Definitive Guide of Solr

Solr权威指南

下卷

兰小伟 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Solr 权威指南 下卷 / 兰小伟著. —北京: 机械工业出版社, 2017.10
(实战)

ISBN 978-7-111-58207-6

I. S… II. 兰… III. 搜索引擎—程序设计—指南 IV. TP391.3-62

中国版本图书馆 CIP 数据核字 (2017) 第 261590 号

Solr 权威指南 下卷

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 何欣阳

责任校对: 李秋荣

印 刷: 北京诚信伟业印刷有限公司

版 次: 2018 年 1 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 20.5

书 号: ISBN 978-7-111-58207-6

定 价: 89.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Preface 序 言

Apache Solr 是使用最广泛的全文检索解决方案，大部分网站都在使用 Solr 来实现搜索功能。然而国内关于 Solr 的资料太少，无奈我只能一点点地啃 Solr 官方提供的 User Guide PDF 文档、Solr Wiki 以及一些纯英文的技术书籍，希望能够借由本书将我学习积累的所有经验倾情传授给那些由于学习 Solr 曲线太陡峭而束手无策的同学们。本书致力于帮助 Java 开发人员更简单、深入地学习 Solr。同时本书还提供了随书源码，其中包含大量可运行的示例代码。本书与随书源码搭配在一起学习会事半功倍！由于目前大数据、云计算的发展如火如荼，各种大数据生态框架如雨后春笋般涌现，给人一种无形的压力。为此，本书也介绍了 Solr 与大数据框架的集成，如果你正好有这方面的需求，希望本书能够给你带来帮助。

为什么写这本书

转眼间，我已经跌跌撞撞走过了 5 个年头，由起初的那个 Java 迷途小书童变身为程序员届的一根老油条，不由感慨万千。由于深谙一个非高校毕业的“正规军”一路走来有多么的艰辛，因此我一直秉持爱开源、爱分享的个性。这么多年来帮助过的程序员太多太多，本着一颗乐于助人的心，我不想大家重走我的弯路。从 2015 年 3 月中旬开始，我在 ITEye 技术社区发布与 Lucene 和 Solr 相关的技术博客，深受大家喜爱。每天联系、咨询我问题的网友越来越多。疲于应付的我，开始意识到仅靠一个人这样一对一地指导是行不通的。而且刚好 Solr 这方面的中文技术书籍在中国还是一片空白，于是萌生了写一本 Solr 中文书籍的想法，希望能够帮助更多的 Solr 技术爱好者。

2015 年 8 月我联系到了华章的杨福川，向他提出了写这本书的想法，得到了他的大力支持。我深知自己过往没有显赫耀眼的工作经历，在一些前辈面前还只是一个晚辈。因此，在创作本书的过程中，查阅了 Solr 官网提供的 Apache-Solr-Ref-Guide、Solr Wiki，并通读了《Solr in action》《Apache Solr 4 Cookbook》《Apache Solr Essentials》《Apache Solr High

Performance》等英文技术书籍。为了能够编写 Solr 与大数据集成相关章节，我又耗费了大量时间通读了《Apache Flume Distributed Log Collection for Hadoop》《Hadoop in Action》《HBase in Action》《Learning Spark》等大数据相关的英文技术书籍。写作本书的过程也成为本人学习提升的过程，为此我花费了整整 1 年的时间。资历尚浅仍可以通过自身努力来弥补，所以我时时刻刻以严谨缜密的态度对待写进书里的每一段文字，除了怀揣着对技术的一种敬畏之情，我知道我还必须为读者负责。

然而造化弄人，在 2016 年的 2 月份，我的颈部莫名其妙长了一个肿瘤，这严重影响了我的身心健康。由于辗转于北京协和医院、解放军总医院等地投医救治，所以这本书的编写工作不得不临时中断。还好我没有放弃，于是在修养了半年之后，又进入了“挑灯夜战”的状态，开始以夜继日地赶稿子。因为已经立下了写书的豪言壮志，所以再苦再累我也是要写完的！由于生病，当初所在的公司要求我立即停薪修养，在看尽了世态炎凉之后，我毅然选择了辞职，打算专职将这本书写好，给读者一个交代。没有了经济来源，只靠自己多年来的积蓄维持生活。我顶着巨大的压力，在大病初愈的情况下，决定倾注全部精力打造这本书。很庆幸我坚持下来了。每天叫醒我的不是闹钟不是鸡汤，也不是其他竞争对手，而是我的决心，因为父母已两鬓白发，快要三十的我还孑然一身。所以我不能虚度光阴，需要为了我爱的人和爱我的人努力奋斗，从而改善他们的生活。这本书也算是给自己 30 岁生日提前备下的一份礼物，并借以纪念不悔的青春岁月。我知道和我有着类似经历的同学太多太多，因此希望这本书能够为学习 Solr 的你们带来帮助和鼓励：定好一个 Target，就永远不要放弃！

准备工作

随书提供了大量的示例代码（本书随书示例源码下载地址：<https://github.com/yida-lxw/solr-book>），其中涉及 MongoDB、ZooKeeper、Hadoop、HBase、Flume、Kafka、Storm、Spark、Scala 等知识点，不仅限于 Solr，所以对于 Java 初学者而言会有一定压力。尽管书中提供了部分大数据框架的集群搭建步骤，但是由于篇幅的限制不可能面面俱到，你还是需要另外查阅其他相关书籍或资料来补充大数据这方面的知识。由于随书源码是基于 Maven 构建的，因此你还需要掌握 Maven 的基本使用方法。为了尽最大努力满足大部分用户的需求，所以从第 14 章开始我将以 Solr 6.2.1 版本为例进行讲解，而 Solr6.x 是要求 JDK 1.8+ 版本的，那么在学习本书之前，你需要提前安装好 JDK 1.7 和 JDK 1.8。如果你有将 Solr 部署在 Tomcat 下的需求，那么你还应安装 Tomcat 环境。对于企业而言，SolrCloud 集群通常会部署在 Linux 环境下，因此本书 SolrCloud 部分是以 CentOS 6.5 为例进行讲解的，或许你还需要掌握 Linux 操作系统的基础知识以及一些 Linux 的常用命令。另外，由于 Solr 是基于 Lucene 构建的，

因此你最好拥有一定的 Lucene 基础再来学习本书内容会感觉更轻松。因为本书自始至终是由浅入深的原则进行编写的, 尽量细致入微地讲解每一步。当然, Solr 源码是使用 Java 编写的, 这也要求你能够熟练掌握 Java 编程语言的知识, 并拥有良好的编码基本功以及编程悟性。而 Solr 中的数据往往来自于关系型数据库, 因此你最好是对关系型数据库有一定的了解。

如何阅读本书

全书分为上下两卷, 总共 16 章, 涵盖了 Solr 各个方面的知识点。本书从前到后按内容的难易程度以循序渐进的方式呈现出来。因此你只需要拥有足够的毅力将它阅读完, 当然最好是能够边读边上机实践, 就可以掌握 Solr。此外每章之间都是相互独立的, 如果你对于某章的内容已经非常熟悉, 那么可以直接跳过选择感兴趣的章节进行学习。当然还是建议大家能够通读本书, 系统学习 Solr, 这样才会对 Solr 有一个更完整的理解, 为你日后从事 Solr 相关的开发工作打下夯实的基础。本书每章开头部分都列举了该章的主要知识点, 可以让你快速了解本章能够学习到的内容。虽然本书中演示的示例代码在随书源码中都可以找到, 但是我还是建议大家能够实际动手去敲一遍, 毕竟只有亲手实践过, 才能将遇到的各种问题真正悟透并彻底解决。这个过程虽是艰辛的, 但也是深刻的, 因为解决问题对于程序员来说就是积累经验的机会。

面向的读者

- ☐ Java 开发工程师;
- ☐ 架构师;
- ☐ Solr 技术爱好者;
- ☐ 各大高校或 IT 培训机构的学弟学妹们。

勘误与反馈

在编写本书的过程中, 尽管我倾注了大量时间与精力, 但是由于水平有限, 书中难免会存在不足与疏漏之处, 还请大家多多批评指正。如果你在阅读本书过程中有任何疑问或者建议需要向我反馈, 可直接发送 E-mail 至 736031305@qq.com 或者添加个人微信 (13476669029) 联系我。

致谢

不知道你拿到这本书的时间是哪一年哪一个季节, 但是对我来说, 这都是我在自己 30

岁之前完成的一个最大的心愿。这是国内真真正正全面介绍 Solr 技术的第一本中文书籍，很开心我做到了。

想感谢的人很多，首先要谢谢爸妈，在我生病期间无微不至地照顾我，并无条件地支持我。

谢谢一路以来理解并鼓励我的朋友和粉丝们，是你们让我不断坚持前行。

谢谢机械工业出版社华章公司的杨福川、高婧雅、李艺在这一年当中对我写作的信任与帮助，没有你们辛勤的付出，就不可能有这本书的面市。非常开心和幸运能够与你们共同完成这样一本书籍。

谢谢我的 Java 启蒙老师习晨龙，是您带我进入了 Java 世界，从此我在汲取知识的路上甘之如饴。

谢谢在这么多年的工作中所有帮助过我的同事，我会一直记得你们。

最后需要感谢的还是我自己，感谢曾经的年少轻狂，感谢一直都存在的梦想，对于梦想我从来没有也永远不会放弃。所以如果你还有梦想，为了你爱的人，为了你自己，请永远不要放弃！

Contents 目 录

序 言

第 11 章 Solr 高级查询 1

11.1 Solr 函数查询 2

11.1.1 Function 语法 2

11.1.2 使用函数查询 4

11.1.3 将函数计算值作为“伪域” 返回 5

11.1.4 根据函数进行排序 6

11.1.5 Solr 中的内置函数 7

11.1.6 自定义函数 13

11.2 Solr 地理空间查询 16

11.2.1 Solr 地理空间简单查询 17

11.2.2 Solr 地理空间高级查询 19

11.3 Pivot Facet 29

11.4 Solr Subfacet 31

11.4.1 Subfacet 语法 32

11.4.2 Subfacet 复杂示例 32

11.5 Solr Facet Function 34

11.5.1 聚合函数 35

11.5.2 聚合函数与 Subfacet 结合 35

11.5.3 Solr 中的 Percentile 函数 36

11.6 JSON Facet API 39

11.6.1 JSON Facet API 简介 39

11.6.2 JSON Facet 简单使用 40

11.6.3 Facet 类型 41

11.6.4 JSON Facet 语法 41

11.6.5 Term Facet 42

11.6.6 Query Facet 43

11.6.7 Range Facet 43

11.6.8 Multi-Select Facet 44

11.7 Interval Facet 47

11.8 Hierarchical Facet 48

11.9 Solr Stats 组件 50

11.10 Solr Terms 组件 52

11.11 SolrTerm Vector 组件 54

11.12 Solr Query Elevation 组件 56

11.13 Solr Result Clustering 组件 59

11.14 本章总结 62

第 12 章 Solr 查询进阶篇 63

12.1 Solr 深度分页 63

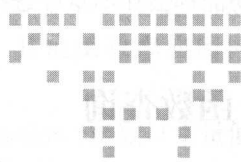
12.2 Solr 自定义排序 66

12.3 Solr Join 查询 70

12.3.1 跨 Core Join	71	13.9.3 加载 Core	119
12.3.2 跨 Document Join	73	13.9.4 交换 Core	119
12.3.3 Block Join	74	13.9.5 重命名 Core	120
12.3.4 Block Join Facet	77	13.9.6 查看 Core 状态	120
12.4 深入 Solr 相关性评分	79	13.9.7 Core 合并	120
12.4.1 Field 权重	79	13.9.8 Core 分裂	121
12.4.2 Term 权重	80	13.10 使用 SolrJ 管理 schema.xml	122
12.4.3 Payload 权重	80	13.10.1 Field 管理	122
12.4.4 Function 权重	81	13.10.2 FieldType 管理	127
12.4.5 邻近 Term 权重	82	13.10.3 Schema 管理	130
12.4.6 Document 权重	83	13.10.4 Schema 管理的事务性批量 操作	132
12.4.7 自定义 Similarity 插件	84	13.11 使用 SolrJ 操作 JSON Request API	133
12.5 Solr NRT 近实时查询	86	13.12 使用 Spring Data Solr	136
12.6 Solr Real-time Get 查询	88	13.12.1 Spring Data Solr 环境 搭建	136
12.7 Solr 评分查询	90	13.12.2 Spring Data Solr 的 CRUD	138
12.8 Solr MoreLikeThis 组件	91	13.12.3 Spring Data Solr 中的 查询	141
12.9 Solr 自定义 Query Parser	95	13.12.4 Spring Data Solr 中的 Repository 详解	143
12.10 本章总结	97	13.12.5 Spring Data Solr 中 Solr- Template 工具类详解	146
第 13 章 SolrJ	98	第 14 章 SolrCloud	153
13.1 什么是 SolrJ	98	14.1 SolrCloud 快速入门	153
13.2 SolrJ 的环境依赖与配置	99	14.2 SolrCloud 工作原理	156
13.3 SolrClient 介绍	101	14.2.1 SolrCloud 的核心概念	156
13.4 SolrJ 简单使用	103	14.2.2 SolrCloud 中的 Shard	157
13.5 SolrJ 查询	106		
13.6 使用 SolrJ 高效导出数据	110		
13.7 SolrJ 增量更新	111		
13.8 SolrJ 原子更新	112		
13.9 使用 SolrJ 管理 Core	116		
13.9.1 创建 Core	117		
13.9.2 卸载 Core	118		

14.2.3	Collection VS Core	158	14.7	SolrCloud 分布式查询	207
14.2.4	索引文档路由	161	14.8	SolrCloud Collection API	208
14.2.5	Shard 的几种状态	162	14.8.1	Collection 常用操作 API	209
14.2.6	Replica 的几种状态	162	14.8.2	Shard 常用操作 API	212
14.2.7	Shard 分割	163	14.8.3	Replica 常用操作 API	215
14.2.8	SolrCloud 里的自动提交	163	14.8.4	集群管理 API	216
14.2.9	SolrCloud 的分布式查询 请求	164	14.9	Solr 索引主从复制	217
14.2.10	读写端的自动容错	171	14.9.1	索引复制简介	217
14.2.11	Zookeeper	173	14.9.2	索引复制的术语	218
14.3	SolrCloud 集群搭建	182	14.9.3	索引复制的配置	219
14.3.1	在 Tomcat 容器下搭建 SolrCloud 集群	183	14.9.4	配置索引复制中继器	221
14.3.2	在 Jetty 容器下搭建 SolrCloud 集群	189	14.9.5	索引复制工作机制	222
14.4	SolrCloud 的基本操作	194	14.9.6	ReplicationHandler HTTP 接口	223
14.4.1	Solr 环境变量设置	194	14.10	跨数据中心的索引复制 (CDCR)	224
14.4.2	创建 Collection	195	14.10.1	什么是 CDCR	224
14.4.3	删除 Collection	196	14.10.2	CDCR 的 Push 机制	225
14.4.4	启动 Solr	196	14.10.3	CDCR 搭建	226
14.4.5	停止 Solr	197	14.10.4	CDCR 配置详解	228
14.4.6	查看 Solr 状态	198	14.10.5	CDCR 的 HTTP 接口	229
14.4.7	Collection 健康检测	198	14.10.6	CDCR 存在的限制	229
14.4.8	管理 Zookeeper 上的配置 文件	199	14.11	本章总结	230
14.5	SolrCloud 配置详解	201	第 15 章 Solr 性能优化	231	
14.5.1	solr.xml 详解	201	15.1	Schema 设计的注意事项	232
14.5.2	zoo.cfg 详解	204	15.2	Solr 索引更新与提交的优化 建议	233
14.6	SolrCloud 分布式索引	205	15.3	索引合并性能调优	234
14.6.1	添加索引文档到 SolrCloud	205	15.4	索引优化的注意事项	235
14.6.2	SolrCloud 里的近实时查询	206	15.5	Solr 缓存	235

15.5.1 Solr 缓存的常见配置参数	236	16.7 SolrCloud 模式下使用 Canal 增量更新索引	264
15.5.2 Filter 缓存	236	16.8 Solr 与 MapReduce 集成	270
15.5.3 Document 缓存	237	16.9 Solr 使用 HDFS 存储索引	271
15.5.4 QueryResult 缓存	237	16.10 使用 Flume 收集数据并索引至 Solr	273
15.5.5 FieldValue 缓存	237	16.11 使用 Solr 实现 HBase 的二级索引	277
15.5.6 HTTP 缓存	238	16.12 Solr 与 Kafka、Flume 集成	282
15.5.7 缓存相关的其他配置	238	16.13 使用 Storm 索引数据至 Solr	286
15.6 Solr 查询性能的优化建议	239	16.14 Spark 与 Solr 进行数据交互	291
15.7 JVM 以及 Web 容器的优化	242	16.15 Solr6 中的 SQL 接口	297
15.8 操作系统级别的优化建议	249	16.15.1 Solr SQL 架构	297
15.9 本章总结	250	16.15.2 Solr SQL 配置	299
第 16 章 Solr 扩展篇	251	16.15.3 发送 Solr SQL 请求	300
16.1 Solr 如何版本升级	251	16.15.4 Solr SQL 语法	301
16.2 Solr 中的伪域	253	16.15.5 Solr SQL 客户端可视化工具的使用	302
16.3 Solr 多语种索引支持	255	16.16 Solr6 中的 Streaming 表达式	304
16.4 Solr 中自定义 Redis 缓存	257	16.16.1 Streaming 语言基础	304
16.5 Solr 如何开启 HTTPS	258	16.16.2 Streaming 源函数	305
16.6 Solr 安全认证	260	16.16.3 Streaming 装饰函数	307
16.6.1 基础安全认证插件	260	16.17 Solr 常见问题解答	310
16.6.2 Solr 中的 Authorization API	263		



第 11 章 Chapter 11

Solr 高级查询

通过第 11 章，你将可以学习到以下内容：

- ❑ 掌握如何使用 Function Query 以及如何自定义 Function Query；
- ❑ 掌握如何使用 Geospatial Query；
- ❑ 掌握如何使用 Pivot Facet 和 Subfacet；
- ❑ 掌握如何使用 JSON Facet API 来实现复杂的数据统计查询；
- ❑ 掌握如何使用 Solr 中的其他查询组件，比如 Elevation（竞价排名组件）；
- ❑ 掌握如何使用 Solr 中的 Result Clustering 组件实现自动结果集聚类分组。

Solr 作为一个强大的文本搜索平台，能够根据输入关键字查询并返回索引文档，你可能也已经了解到了 Solr 的一些核心功能，比如文本分词、关键字高亮、结果集分组等。尽管对于大多数搜索程序来说，将那些与用户查询最佳匹配的索引文档返回给用户是非常重要的，但是 Solr 还有另外一个比较常见的使用场景：聚集结果集用于数据统计分析。Solr 的 Pivot Facet 支持叠加统计多个 Facet（维度），它能够在单个查询中对任意的聚合分类进行计算统计，这使得 Solr 在提供数据分析报告方面变得很有用并且还十分高效。Solr 另一个核心功能就是在查询时能够对数据执行一个 Function（函数）进行动态计算，函数计算后的结果可以被用于 Filter Query、文档的相关性评分、文档的排序、作为文档的“伪域”被返回。Solr 还提供了强大的 Geospatial（地理空间）查询功能，Geospatial 查询允许你根据一个点或者一个区域进行多边形查询，或以经纬度为圆心在指定半径的圆内进行查询，实现附近的位置查询（比如查询当前用户所处位置附近的酒店或饭店）。有时候，你期望在返回的索引文档的域中引用外部数据源，Solr 提供了这个功能。Solr 还支持在同一个 Solr 实例内跨 Core 在一个外键域上执行 Join 操作，这类似于 SQL 里的两个表根据外键进行多表连接查询。上

述每个复杂的功能都会在本章中进行讲解。

11.1 Solr 函数查询

Solr 中的 Function Query (函数查询) 允许你为每个索引文档执行一个函数进行动态计算值。Function Query 是一个比较特殊的查询, 函数动态计算后得到的值可以作为一个关键字添加到查询中, 也可以作为文档的评分, 就像是一个普通的關鍵字查询同时还能生成相关性评分。通过使用 Function Query, 函数动态计算值可以被用于修改索引文档的相关性评分, 以及查询结果集排序, 而且函数动态计算值还可以作为一个“伪域”被动态添加到每个匹配的索引文档中并返回给用户。Function Query 还支持嵌套, 意思就是一个 Function 的输出可以作为另一个 Function 的输入, Function 支持任意深度的嵌套。

11.1.1 Function 语法

Solr 中标准的 Function 语法是先指定一个 Function 名称, 后面紧跟着一对小括号, 小括号内可以传入零个或多个输入参数, 语法使用示例如下:

```
functionName()
functionName(input1)
functionName(input1, input2)
functionName(input1, input2, ..., inputN)
```

Function 的输入参数可以是以下任意一种形式:

❑ 一个常量值 (数字或者字符串), 语法:

```
100, 1.45, "hello world"
```

❑ 一个域名称, 语法:

```
fieldName, field(fieldName)
```

❑ 另外一个 Function, 语法:

```
functionName(...)
```

❑ 一个变量, 语法:

```
q={!func}min($f1,$f2)&f1=sqrt(popularity)&f2=1
```

尽管 Solr Function 乍一看让人有点不知所措, 其实 Solr 文档中定义了每个 Function 的输入参数的类型, 大部分的 Function 都遵循 Function 的标准语法, 但是 Constant Function (常量函数)、Field Function (域函数)、Parameter Substitution (替换变量) 这些属于特例, 它们支持另一种简单语法。Constant Function (常量函数) 的语法就是值本身。Field Function (域函数) 的语法就是域的名称被一个名称为“field”的函数包裹。Parameter Substitution

(替换变量)的语法就是函数的输入变量使用的是一个 \$ 开头的变量,该变量引用自请求 URL 的查询文本中定义的变量。除了这 3 个特例,其他函数都使用标准的 Function 语法。

因为 Function 的所有输入参数可以被看作是一个 Function (函数)(常量值可以被看作常量函数),所以 Function 的标准语法在概念上来讲,就可以理解为 functionName(function1, ..., functionN)。假设索引文档中有个 fieldContainingNumber 域,它其中有个值为 -99,那么请思考下面几个 Function 的使用示例:

```
max(2, fieldContainingNumber)           // 输出结果: 2
max(fieldContainingNumber, 2)           // 输出结果: 2
max(2, -99)                             // 输出结果: 2
max(-99, 2)                             // 输出结果: 2
max(2, field(fieldContainingNumber))    // 输出结果: 2
max(field(fieldContainingNumber), add(1,1)) // 输出结果: 2
```

从上面示例你会注意到,你可以为 Constant Function 常量函数(甚至你可以为其他任意标准函数)使用 Field Function 进行包装,尽管输入的参数顺序以及每个输入参数的含义会有所不同,但是它们最终都是用于计算 -99 和 2 之间的最大值。将一个函数的输入参数看作另外一个函数的好处就是它允许你用任意的嵌套函数来实现复杂计算。并不是所有的 Function (函数)都支持同样类型的输入参数,有些 Function (函数)期望接收字符串类型常量参数,而另外一些 Function (函数)可能期望接收 Integer 或者 Float 类型的数字。假设 fieldContainingString 域的域值为 "hallo",请思考下面的函数调用示例:

```
strdist("hello", fieldContainingString, edit) // 输出结果: 0.8
strdist("hello", "hallo", "edit")           // 输出结果: 0.8
```

strdist 函数用于计算两个字符串之间的相似度,相似度计算是基于一个指定的算法进行计算,使用哪种算法是通过函数的第 3 个参数进行指定,我们示例中的 "edit" 表示采用编辑距算法。假如我们将参数的数据类型指定为错误的,函数将会返回什么呢?

```
strdist("hello", 1000, edit)              // 输出结果: 0
strdist(1000, "1000", edit)               // 输出结果: 1
strdist("1001", 1000, edit)               // 输出结果: 0.75
```

你可能会觉得函数会抛出异常,然而实际上函数内部会适当地自动进行数据类型转换,比如在示例中,将数字常量 1000 转换成字符串 "1000"。在大多数情况下,你并不能安全地将一个字符串转换成一个数字,此时 Solr 可能会抛出一个异常。因此,需要谨记:函数嵌套确实很好用,但是并不是所有的函数都可以随意嵌套,你需要考虑每个函数的输入参数类型是否正确。

Solr 的 Function 可以影响相关性评分,可以被用于 Filter Query 过滤结果集,可以基于函数计算值进行排序,可以将函数计算值作为索引文档的“伪域”并返回,甚至可以基于函数计算值进行 Facet 查询统计。下一节我们将深入学习这些用法。

11.1.2 使用函数查询

为了便于后续的示例讲解，请大家从随书源码中获取 Core 的相关配置文件、测试数据及导入测试数据的测试类，根据我们前面章节所学的知识将本节测试环境需要的 "news" Core 搭建好。

在 Solr 中执行一个典型的关键字查询，需要在倒排索引中查找关键字，同时计算每个匹配索引文档的相关性评分，从而决定哪些索引文档与查询关键字比较相关，最后作为结果集返回。然而查询不仅能基于搜索关键字，你可以在查询中插入一个 Function 并将其看作另外一个搜索关键字。为了演示 Function Query，请建立 "news" Core 并运行随书源码中的 IndexNews 类导入测试数据。假如已经成功导入了测试数据，可以执行下面的查询示例：

```
http://localhost:8080/solr/news/select?
q="United States" AND France AND President AND _val_:"recip(ms(date),1,100,100)
"&indent=true
```

上面的查询表示查询包含 "United States" 短语且包含 France 和 President 关键字，并且函数计算值在 [1, 100] 区间范围内的索引文档。这里有 3 个关键点需要引起你的注意：

- `_val_` 语法：用于注入一个 Function Query，这里的 `_val_` 可以看作主查询中的一个查询 Term。
- Function Query 并不会改变最终返回结果集中索引文档的总数。
- 查询的最后相关性评分一般是查询中每个 Term 的相关性评分的总和，"United States"、France 和 President 这些 Term 的相关性评分是基于 tf-idf 相似度算法进行计算的，但是 Function Query 的评分计算是函数自身的计算值。

基于上述 3 点，你可以了解到示例中的 Function Query 是为了给新添加的索引文档进行加权。最新的索引文档的相关性评分可能是 100，而最旧的索引文档的评分可能是 1，剩下的索引文档的评分会落在 [1, 100] 之间。注意，每个索引文档的最后评分是标准化的，这意味着每个索引文档的最后评分不会都到达 100 分，最近添加的索引文档相比之前显示会更靠前。

Function 在 Solr 中无处不在，它可以对用户的 `q` 参数进行加权，它还可以在不同的 Query Parser 中使用，比如在 eDisMax Query Parser 中通过 `bf` 参数指定 Function；它还可以作为 Filter Query 的一部分，用于索引文档排序等。但是最重要的是你需要了解 Function Query 是如何被执行的。前面的示例中你已经见过了 `"_val_"` 这样的语法，你可能还记得我们之前介绍过的 Function Query Parser，可以通过一个本地参数 `!func` 来构造一个 Function Query，比如：`{!func}functionName(…)`。Function Query 本质就是将函数计算值作为构造的函数查询的评分，因此，以下几种查询语法是等价的：

```
q=solr AND _val_:"add(1, boostField)"
q=solr AND _query_:"{!func}add(1, boostField)"
q=solr AND {!func v="add(1, boostField)"}
```


为一个查询，添加一个 Function 看起来非常有用，它能修改查询匹配的索引文档的评分。如果你期望过滤掉函数计算值不在指定范围内的索引文档，可以使用 Function Range Query Parser 来解决函数计算值范围过滤。

如果你需要根据函数计算值的范围来过滤索引文档，那么 Function Range Query Parser (简称 frange) 会比较适用你的使用场景，Frange 过滤器通过执行一个指定的 Function Query，过滤掉函数计算值不在指定范围内的索引文档。为了演示这种功能，我们搭建测试环境。这里会用到随书源码中的 "salestax" Core，Core 相关配置文件和测试数据以及导入数据测试类请从相应章节中查找获取。导入完成之后，请看下面这个查询示例：

```
http://localhost:8080/solr/salestax/select?q=*:*&
fq={!frange l=10 u=15}product(basePrice, sum(1, $userSalesTax))&
userSalesTax=0.07
```

以上查询先通过 sum 函数计算 \$userSalesTax 和 1 的价格之和，然后将 basePrice 域的域值与 sum 函数计算返回值通过 product 函数求乘积，最后通过 frange 过滤器的 l (即 lower 表示最小值) 和 u (即 upper 表示最大值) 参数定义了 product 函数计算返回值的取值范围，符合这个区间范围限制的索引文档将会被返回。你还可以设置 incll (即 include lower) 和 inclu (include upper) 参数来指定是否包含两个边界值。

你可能会说，能不能自定义一个 Function 来灵活地过滤任意查询匹配的结果集？关于自定义 Function 相关内容会在本章的后续章节讲解。现在你已经知道如何为查询添加 Function，并且理解了函数评分是如何计算的，接下来让我们继续使用函数动态计算值来代替静态的域值。

11.1.3 将函数计算值作为“伪域”返回

在上一节，你已经了解到函数的输入参数可以被看作是一个函数，既然如此，那么似乎我们可以使用 Function 来替换 Field，因为 Field 和 Function 最终都是返回一个值。事实也是如此，你不仅可以为每个索引文档动态计算得到一个数值，还可以将这个数值作为一个“伪域”随索引文档一起返回。重新回到我们在上一节中的 "salestax" 示例，执行下面这个查询：

```
http://localhost:8080/solr/salestax/select?q=*:*&
userSalesTax=0.07&
fl=id,basePrice,product(basePrice, sum(1, $userSalesTax))
```

上面这个查询返回的结果会是怎样的呢？正如你看到的那样，你会发现返回的索引文档中多了一个“伪域”，“伪域”的域名称就是我们定义的函数表达式，“伪域”的域值就是函数表达式最终的计算值。之所以称为“伪域”，是因为它并不真正存在于我们的索引数据中，但是它仍然会像其他存储域一样被一起返回。“伪域”的名称使用函数表达式可能会显得冗长难看，不过值得庆幸的是，Solr 提供了为“伪域”定义任意你想要的别名的功能，具

体如何为“伪域”定义别名，请看下面这个示例：

```
http://localhost:8080/solr/salestax/select?q=*:*&
userSalesTax=0.07&
fl=id,basePrice,totalPrice:product(basePrice, sum(1, $userSalesTax))
```

这里我们为 "product(basePrice, sum(1, \$userSalesTax))" 这个伪域定义了一个别名 `totalPrice`，最终返回结果里伪域名称就是我们这里定义的别名了。正因为你可以为“伪域”定义任意的别名，因此也就意味着可以将“伪域”的别名定义为索引文档中真实存在的域的域名称，这样就可以直接使用“伪域”的值来冒充该域的真实域值。当你期望根据用户权限来控制某些用户没有权限访问某个域的真实域值的时候，通过“伪域”别名来冒充真实域会对你很有用。

通过使用 `Function`，你可以在域的域值返回之前对其进行任意操纵，比如经过函数计算变换它的值。你不仅可以通过函数修改文档中任何域的域值，还可以通过函数修改文档的相关性评分，从而影响文档是否应该被返回，或者文档在返回的结果集中的排序。

11.1.4 根据函数进行排序

在上一节，你了解了如何将一个函数的动态计算值添加到索引文档中作为一个“伪域”在查询结果集中返回；你也知道了如何根据函数计算值来对查询结果集进行过滤，以及如何使用函数来修改匹配文档的相关性评分。接下来，让我们继续学习如何基于函数动态计算值来对查询结果集进行排序。根据函数动态计算值来对查询结果集进行排序的语法与普通查询中根据某个域排序的语法没什么太大的不同，具体请看下面这个查询示例：

```
http://localhost:8080/solr/salestax/select?q=*:*&
userSalesTax=0.07&
sort=product(basePrice, sum(1, $userSalesTax)) asc, score desc
```

上面这个查询，根据 `product` 函数计算值升序进行排序，然后再按文档的评分降序排序，你可以结合其他函数构造更为复杂的 `Function Query`，比如：

```
http://localhost:8080/solr/salestax/select?
q=_query_:"{!func}recip(ms(date),1,100,100)"&
userSalesTax=0.07&
totalPriceFunc=product(basePrice, sum(1, $userSalesTax))&
fq={!frange l=10 u=15 v=$totalPriceFunc}&
fl=*,totalPrice:$totalPriceFunc&
sort=$totalPriceFunc asc, score desc
```

上面这个查询首先根据 `Function Query Parser` 对 "{!func}recip(ms(date),1,100,100)" 查询表达式进行解析，构造成 `Function Query`；然后通过 `_query_` 语法将 `Function Query` 转换成普通的 `Query`，转换后的查询并没有过滤任何索引文档，它只是用来根据文档的 `date` 域的时间远近对索引文档进行加权；然后通过 `fq` 对 `$totalPriceFunc` 变量表示的函数最终计算

值进行区间范围过滤,不在 [10, 15] 区间内的索引文档将会被过滤掉,通过 sort 参数先按照 \$totalPriceFunc 变量表示的函数计算值进行升序排序;再按照索引文档的评分降序排序;最后通过 fl 里将函数计算值当作索引文档的“伪域”一并返回。这个示例综合使用了我们前面所讲解的知识点。

11.1.5 Solr 中的内置函数

到目前为止,你已经知道如何在 Solr 中应用 Function。由于 Solr 内置的函数非常多,而且还在不断增加中,所以本书这部分内容不可能面面俱到,如果本书有遗漏某个函数没有提及,读者可以自行查阅资料学习。但是我会尽量覆盖 Solr 中内置的大部分常用函数,并详细解释每个函数的用途以及使用语法。Solr 中的内置函数大致分为 5 类: data transformation (数据转换)、Math (数学计算)、Relevancy (计算相关性评分)、Distance (距离计算)、Boolean (布尔操作)。

1. 数据转换类函数

Solr 中比较常用的函数大都是转换类函数,即将数据通过一个或多个函数计算从一个值转换成另一个值。下面会详细介绍每个转换类函数的用途和用法(如表 11-1 所示)。

表 11-1 数据转换类函数表

函 数	描 述
def(x, y)	如果 x 值不存在就返回 y, 否则就返回 x, 一般用于设置默认值, 防止出现空值情况
field(fieldName)	返回指定索引域(即该域必须 indexed=true)的域值, 但是请注意, field 函数不适用于多值域, 即指定域的 multiValued 不能为 true。虽然你对多值域执行此函数并不会抛出异常。如果指定域是 TrieDateField 类型, 那么返回该 date 域的 toString 形式, 如果指定域是 DateField 或 LegacyDateField 类型, 那么会直接返回 "ord" 字符串, 如果指定域的域值不存在会返回 0, 如果指定域是 boolean 域, 那么直接返回 true 或者 false, 如果是数字域, 那么直接返回该域的域值, 如果该数字域的域值不存在, 那么会返回默认值零
map(x, min, max, target) map(x, min, max, target, else)	map(x, min, max, target) map(x, min, max, target, else) 当 x 的值在 [min, max] 范围内, 则输出 target, 当 x 的值不在 [min, max] 范围内, 此时分两种情况, 当 else 参数有指定, 则输出 else, 如果 else 参数未指定, 则输出 x。如果 x 是空值, 会返回默认值零
ms(time2, time1) ms(time1) ms()	返回 time2 与 time1 的差值(毫秒), 这里的毫秒数指的是距离 1970 年 1 月 1 日零点零分零秒这个时间点以来的毫秒数。如果 time2 未指定, 那么直接返回 time1, 如果 time1 也未指定, 那么返回当前时间距离 1970-01-01 00:00:00 的毫秒数。注意, 这里的 time1, time2 必须都是 date 域
ord(fieldName)	返回 Term 在索引中的位置(从 1 开始计算), 默认按照字符串的 ASCII 码进行排序确定先后位置。因此传入的域必须是 indexed=true 的字符串域, 并且 ord 函数不支持多值域, 即域的 multiValued 不能为 true。注意: ord 函数依赖于 Term 在索引中的位置, 因此当你添加或删除索引文档, 或者当你使用了 MultiSearcher 时, ord 函数的返回值都会随之改变

(续)

函 数	描 述
<code>rord(fieldName)</code>	与 <code>ord</code> 函数类似, 只不过 <code>rord</code> 函数是 <code>ord</code> 函数的倒序
<code>scale(x, num1, num2)</code>	将 x 限制在 $[num1, num2]$ 区间内, 如果 x 是 <code>date</code> 域, 内部会自动将 <code>date</code> 转成毫秒数, 如果 x 是字符串域, 那么会抛异常。如果某个索引文档被删除或者域值为空, 此时会返回默认值 0.0, 这也就意味着 <code>scale</code> 函数无法区分文档是被删除了还是域值为空, 也就是说即便域值都大于 0, 也有可能被映射为 0.0, 此时你需要使用 <code>map</code> 函数来将 0.0 映射到正确的范围内。比如 <code>scale(map(x,0,0,5),1,2)</code>
<code>top(x)</code>	强制 x 值必须从包含所有索引的顶级 <code>IndexReader</code> 中获取, 而不是从每个段文件的 <code>IndexReader</code> 获取。比如, 一个域值在单个段文件中的顺序与在整个索引中的位置是不同的。 <code>ord</code> 和 <code>rord</code> 函数内部会隐式的调用 <code>top</code> 函数, 这也就意味着 <code>ord(x)</code> 等价于 <code>top(ord(x))</code>
<code>literal(val)</code>	将 <code>val</code> 以字符串形式返回, <code>val</code> 两头可以加单引号、双引号, 你不加单引号且不加双引号也可以。如果你传入的是数字, 会自动将数字转换字符串。比如: <code>literal(100)</code> 会返回 "100", <code>literal(price)</code> 或 <code>literal("price")</code> 或 <code>literal('price')</code> 都会返回 "price"
<code>currency(field_name, [CODE])</code>	<code>currency</code> 函数用于货币汇率转换, 如果你未指定货币代号即 <code>[CODE]</code> 参数时, 那就按照默认的货币进行转换。 <code>currency</code> 函数使用示例: <code>currency(price_c)</code> <code>currency(price_c,EUR)</code> 具体请查阅 <code>CurrencyField</code> 类

2. 数学函数

数学计算是比较常用的数据分析操作。Solr 全面支持数学计算, 支持包括加减乘除以及三角函数等多种数学函数。表 11-2 列举了 Solr 中支持的数学函数。

表 11-2 Solr 中支持的数学函数

函 数	描 述
<code>abs(x)</code>	返回 x 的绝对值
<code>acos(x)</code>	返回 x 的反余弦值
<code>asin(x)</code>	返回 x 的正弦值
<code>atan(x)</code>	返回 x 的反正切值
<code>atan2(x, y)</code>	返回原点至点 (x, y) 的方位角, 即与 x 轴的夹角, 返回值的单位为弧度, 取值范围为 $(-\pi, \pi]$
<code>cbrt(x)</code>	返回 x 的立方根
<code>ceil(x)</code>	返回大于或者等于 x 的最小整数
<code>cos(x)</code>	返回 x 的余弦值
<code>cosh(x)</code>	返回 x 的双曲余弦值
<code>deg(x)</code>	将弧度 x 转换成角度

(续)

函 数	描 述
div(x, y)	返回 $x \div y$
e()	基于自然对数返回一个近似欧拉数
exp(x)	返回 e 的 x 次幂
floor(x)	返回大于或者等于 x 的最大整数
hypo(x, y)	返回一个直角的斜边: $\sqrt{x^2 + y^2}$
linear(m, x, b)	返回线性函数 $f(x) = m * x + b$ 的函数值
ln(x)	返回 x 的自然对数
log(x)	返回以 10 为底数, x 的对数
pi()	返回常量 π
pow(x, y)	返回 x 的 y 次幂
product(x, ... n) mul (x, ... n)	返回多个数字的乘积
rad(x)	将角度 x 转换成弧度, 与 $\deg(x)$ 函数相对应
recip(x, m, a, b)	返回倒数函数 $a/(m * x + b)$ 的函数值
rint(x)	返回离 x 最近的整数
sin(x)	返回 x 的正弦值
sinh(x)	返回 x 的双曲正弦值
sqrt(x)	返回 x 的平方根
sub(x, y)	返回 $x - y$
sum(x, ... n) add (x, ... n)	返回多个数字的和
tan(x)	返回 x 的正切值
tanh(x)	返回 x 的双曲正切值

3. 相关性评分函数

Solr 的相关性评分默认是基于 DefaultSimilarity 类进行计算而来的, DefaultSimilarity 利用索引的统计信息来决定哪些索引文档与查询匹配。这些相关性评分是针对每个索引文档进行计算得到一个综合的评分, 你还可以使用相关性评分函数对个别查询进行部分评分 (比如你只想返回 Term 的出现频率信息)。相关性评分的所有核心统计都包含于相关性评分函数中, 比如 tf-idf。表 11-3 列举了 Solr 中支持的相关性评分函数。

表 11-3 Solr 中支持的相关性评分函数

函 数	描 述
docfreq(fieldName, term)	返回 fieldName 域上包含指定 term 的索引文档的总个数
idf(fieldName, term)	返回 fieldName 域上出现 term 的 document 的频率

(续)

函 数	描 述
maxdoc()	返回索引中文档总个数，包含已经标记为删除但是尚未写入到磁盘的索引文档
norm(fieldName)	返回指定 fieldName 域存储在索引中的 norm 值
numdocs()	返回索引中的文档总格式，不包含已经标记为删除尚未写入到磁盘的索引文档
query(subquery, default)	<p>为给定的 subquery 计算评分，如果 subquery 没有匹配任何索引文档，那么返回 default 作为 subquery 的评分。subquery 可以是 \$ 参数指代的查询表达式，也可以是本地参数 v 指定的查询表达式，比如：query({!v="content:(solr OR lucene)"}))。</p> <p>以下是几个 query 函数的使用示例：</p> <p>q=product(popularity, query({!dismax qf=text v='solr rocks'}))</p> <p>q=product(popularity, query(\$qq))&qq={!dismax qf=text}solr rocks</p> <p>q=product(popularity, query(\$qq, 0.1))&qq={!dismax qf=text}solr rocks</p>
sumtotaltermfreq(fieldName)	返回 fieldName 域在索引中的被索引的 token 总个数
sttf(fieldName)	
termfreq(fieldName, term)	返回索引中 fieldName 域上 term 的出现频率
tf(fieldName, term)	返回 term 在 Document 中的出现频率，默认实现是对 termfreq() 开平方根
totaltermfreq(fieldName, term)	返回 term 在整个索引中的出现频率
ttf(fieldName, term)	

可以使用上面表中的相关性函数来重写评分计算，请思考下面这个 Solr 查询示例：

```
http://localhost:8080/solr/salestax/select?
fq={!cache=false}text:"microsoft office"&
q={!func}sum(
  product(
    tf(text, "microsoft"),
    idf(text, "microsoft")
  ),
  product(
    tf(text, "office"),
    idf(text, "office")
  )
)
```

上面这个查询首先分别计算 "microsoft" 和 "office" 的 tf 和 idf，然后分别计算 tf 和 idf 的乘积，最后返回两个乘积的和作为文档的最后评分。然后根据短语 "microsoft office" 进行过滤，将不符合 fq 参数要求的索引文档过滤掉。通过这些相关性评分函数，Solr 为你打开了评分模型，可以随心所欲地在查询时通过相关性评分函数来干预文档评分。

4. 距离计算函数

有时候你希望能够计算两个值之间的距离，比如你要计算地球上两个点（甚至两个向量）之间的空间距离，这在地理空间搜索中会比较有用。你还可以计算两个字符串之间的相似度，表 11-4 列举了 Solr 中支持的距离计算函数。

表 11-4 Solr 中支持的距离计算函数

函 数	描 述
<code>dist(power, x1, ..., n1, x2 ... n2)</code>	<p>基于 power 参数指定的距离测量方法计算在 N 维空间里 2 个向量或坐标点之间的距离, power 参数可选值:</p> <p>0—稀疏计算 (0-norm)</p> <p>1—曼哈顿距离 (1-norm)</p> <p>2—欧几里得距离 (2-norm)</p> <p>infinite norm (向量中的最大值)</p> <p>使用示例如下:</p> <p><code>dist(2, x, y, 0, 0)</code>: (0, 0) 和 (x, y) 之间的欧氏距离</p> <p><code>dist(1, x, y, 0, 0)</code>: (0, 0) 和 (x, y) 之间的曼哈顿距离</p> <p><code>dist(2, x, y, z, 0, 0, 0)</code>: (0, 0, 0) 和 (x, y, z) 之间的欧氏距离</p> <p><code>dist(1, x, y, z, e, f, g)</code>: (x, y, z) 和 (e, f, g) 之间的曼哈顿距离</p>
<code>sqdist(x1, ..., n1, x2 ... n2)</code>	计算 dist 函数返回的欧几里得距离的平方, 这种方式很高效, 因为它消除了平方根计算, 如果在排序或者相关性加权时, 只关心最后的相对顺序, 而不关心实际准确距离, 就可以使用此函数
<code>hsin(radiusInKM, isDegrees, x1, y1, x2, y2)</code>	半正矢函数 Haversine 是 Lucene 的 Spatial4j 计算地理空间距离的默认公式, 本质上是球面余弦函数的一个变换
<code>geohash(lat, lon)</code>	<p>传入 lat (纬度) 和 lon (经度) 通过 geohash 算法计算坐标点的字符串编码。</p> <p>在 Solr 中, 地理空间搜索主要是基于 GeoHash 和 Cartesian Tiers 这 2 个算法来实现, 其中 GeoHash 算法可以将经纬度的二维坐标变成一个可排序、可比较的的字符串编码。</p> <p>在编码中的每个字符代表一个区域, 并且前面的字符是后面字符的父区域</p>
<code>ghhsin(radiusInKM, geohash1, geohash2)</code>	对两个 geohash 值执行 hsin 函数, 代替角度和弧度, geohash1 和 geohash2 可以是 geohash 函数的返回值, 也可以是 GeoHashField 域
<code>strdist(s1, s2, distType)</code> <code>strdist(s1, s2, "ngram", ngramSize)</code>	<p>计算字符串的相似度, 返回值取值范围 [0, 1], 返回 1 即表示两个字符串完全一样。distType 参数可选值:</p> <p>jw (Jaro-Winkler Distance 算法, 这是一种计算两个字符串之间相似度的算法)</p> <p>edit (著名的 Levenshtein distance 算法)</p> <p>ngram (NGram Distance 算法)</p> <p>如果你想使用自定义的字符串相似度算法, 可以指定为实现了 StringDistance 接口的实现类的完整包路径</p>
<code>geodist(sfield, lat, lon)</code> <code>geodist(sfield, pt)</code> <code>geodist()</code>	计算地球上两点之间的距离, sfield 参数表示一个 spatial field, 另外一个参数表示坐标

从以上的表你可以了解到, Solr 全方位支持距离计算函数。dist 函数允许你指定 0-norm、1-norm、2-norm、无穷 norm 用于 N 维空间内两个点或向量的距离计算, 比如计算二维空间内两点的欧几里得距离: `dist(2, x1, y1, x2, y2)`, 计算三维空间内两点的曼哈顿距离: `dist(1, x1, y1, z1, x2, y2, z2)`。

sqdist 函数相比欧几里得函数计算执行开销更小, 它返回的是欧几里得距离的平方根, 对于二维空间里的坐标点, 平方欧式距离计算的是勾股定理 ($a^2 + b^2 = c^2$) 中的 c^2 , 因为欧式距离计算必须要额外的对 c^2 进行开平方根从而得到确切的 c 值, 如果你只需要在文档排序

或者文档相关性评分时获取到文档的相对顺序，而不关心两点之间的实际准确距离值，那么使用 `sqdist` 函数计算性能会更高。

`hsin` 函数用于计算球面上两点的距离。`radiusInKm` 参数表示球面的半径，如果你想计算地球上两点的距离，地球的近似半径值是 6371.01（赤道半径），由于地球并不是一个完美的球体，因此这个近似半径值的精确度还可以再提升 0.5%。如果指定的是经纬度，那么 `isDegrees` 需要设置为 `true`，如果坐标点指定的是弧度，那么 `isDegrees` 参数设置为 `false`。`x1`、`y1` 表示第一个坐标点，`x2`、`y2` 表示第二个坐标点。

`ghhsin` 函数与 `hsin` 函数类似，但是 `ghhsin` 函数接收的不是角度和弧度，而是 `geohash` 值。`geohash` 函数可以接收经度和未读值，并将它们进行 `geohash` 编码，得到的编码字符串可以作为 `ghhsin` 函数的输入参数，如果在索引中你的某个域存储的是 `geohash` 函数编码后的字符串，那么可能会用到这些函数。

`strdist` 函数用于计算两个字符串之间的相似度，一般用于相似 Term 的模糊匹配。如果将一个字符串看作一个多字符的向量，`strdist` 函数计算的是两个字符串（字符向量）之间的距离，最终计算得到的相似度值范围是 $[0, 1]$ ，0 表示一点也不相似，1 表示两个字符串完全相等。`strdist` 函数的 `s1`、`s2` 参数表示输入的两个字符串，`distType` 参数用于指定距离计算的算法，如果 `distType` 参数值为 `ngram`，默认使用 2 个字符串相比较，但是可以通过 `ngram` 参数覆盖。

`geodist` 函数用于计算地球上两点之间的空间距离。`sfield` 参数必须是 `LatLonType` 域，函数返回的距离单位是千米，`lon` 参数表示经度，`lat` 表示纬度，`pt` 参数即 `point` 的缩写，即坐标点 (`x`, `y`)。`geodist` 函数内部使用 `hsin` 函数，简化了使用语法。`geodist` 函数是 Solr 中最常用的距离计算函数，我们会在下一章节更详细的讲解它在 `Geospatial Search`（地理空间查询）中的应用。

5. 布尔函数

Boolean 操作不仅可以用于关键字查询，它还可以用于构建或连接任意复杂的 `Function Query`（函数查询），通过 `if`、`and`、`or`、`not`、`xor`、`exists` 等函数，可以检查域值或其他函数计算以及基于这些检查有条件的返回域值，表 11-5 列举了 Solr 支持的布尔函数。

表 11-5 Solr 支持的布尔函数

函 数	描 述
<code>and(x, y)</code>	如果 <code>x</code> 和 <code>y</code> 都是 <code>true</code> ，则最后返回 <code>true</code> ，否则返回 <code>false</code>
<code>exists(x)</code>	判断 <code>x</code> 值是否存在，如果 <code>x</code> 值存在则返回 <code>true</code> ，反之 <code>false</code>
<code>if(x, a, b)</code>	如果 <code>x</code> 为 <code>true</code> ，那么返回 <code>a</code> ，否则返回 <code>b</code> ，类似三目运算
<code>not(x)</code>	如果 <code>x</code> 为 <code>true</code> ，则返回 <code>false</code> ，反之返回 <code>true</code>
<code>or(x, y)</code>	如果 <code>x</code> 或 <code>y</code> 为 <code>true</code> 或者 <code>x</code> 和 <code>y</code> 都为 <code>true</code> ，则最后返回 <code>true</code> ，如果 <code>x</code> 和 <code>y</code> 都为 <code>false</code> ，则最后返回 <code>false</code>
<code>xor(x, y)</code>	即位运算里的异或操作，比如 $0 \wedge 1 = 1$

Solr 提供了这么多丰富的函数供我们选择, 应该能够满足大部分用户的需求, 与 Solr 中的其他功能一样, 你也可以自定义 Function 来扩展 Solr 的 Function, 接下来将学习如何创建我们自己的 Function。

11.1.6 自定义函数

有时候, 你可能想要执行某些数据操作, 但 Solr 内置函数并不支持。值得庆幸的是, 在 Solr 中, 可以很简单地实现自己的自定义函数。可以从技术上进行内存计算做任何事情, 比如访问外部文件或数据源获取数据, 甚至运行任意你想要的代码。实现自定义的 Function 唯一的约束就是你能容忍函数计算完成耗费多长时间。因为自定义的 Function 代码可能会针对每个索引文档进行计算, 它需要在合理的响应时间内快速的作出响应。

在本节, 我们将演示如何创建一个自定义 Function 来将多个域的域值拼接成一个字符串。为了能够在 Solr 中以插件的方式使用我们的自定义 Function, 你需要完成以下 3 个步骤:

- 1) 编写一个类表示你的 Function, 这个类需要继承 ValueSource 类, 它能够索引中的每个索引文档返回一个计算值。
- 2) 编写一个 ValueSourceParser 类, 它能够解析自定义 Function 的语法并将其解析为变量传递给第一步自定义的 ValueSource 类。
- 3) 在 solrconfig.xml 中注册你在第二步中定义的 ValueSourceParser 类, 指定它的完整类路径以及函数名称, 但自定义 Function 执行时, 会使用你定义的函数名称, 并且自定义的 ValueSourceParser 类解析输入参数并传递给第一步自定义的 ValueSource 类。

我们将要实现的 concatenation 函数需要继承 Solr 中的 ValueSource 类, 并重写其 getValues 方法, 最后返回一个 FunctionValues 对象, FunctionValues 对象可以为索引中每个索引文档返回计算值, 以下的代码演示了如何创建一个 ConcatenateFunction 类:

```
/**
 * Created by Lanxiaowei
 * 自定义 Concatenate 函数
 */
public class ConcatenateFunction extends ValueSource {
    protected final ValueSource valueSource1;
    protected final ValueSource valueSource2;
    protected final String delimiter;
    public ConcatenateFunction(ValueSource valueSource1,
                               ValueSource valueSource2,
                               String delimiter) {
        if (valueSource1 == null || valueSource2 == null){
            throw new SolrException(
                SolrException.ErrorCode.BAD_REQUEST,
                "One or more inputs missing for concatenate function"
            );
        }
    }
}
```

```

        this.valueSource1 = valueSource1;
        this.valueSource2 = valueSource2;
        if (delimiter != null){
            this.delimiter = delimiter;
        }
        else{
            this.delimiter = "";
        }
    }

    public FunctionValues getValues(Map context, LeafReaderContext readerContext)
    throws IOException {
        final FunctionValues firstValues = valueSource1.getValues(
            context, readerContext);
        final FunctionValues secondValues = valueSource2.getValues(
            context, readerContext);
        return new StrDocValues(this) {
            @Override
            public String strVal(int doc) {
                return firstValues.strVal(doc)
                    .concat(delimiter)
                    .concat(secondValues.strVal(doc));
            }
            @Override
            public String toString(int doc) {
                StringBuilder sb = new StringBuilder();
                sb.append("concatenate(");
                sb.append("\"" + firstValues.toString(doc) + "\"")
                    .append(',')
                    .append("\"" + secondValues.toString(doc) + "\"")
                    .append(',')
                    .append("\"" + delimiter + "\"");
                sb.append(')');
                return sb.toString();
            }
        };
    }

    @Override
    public boolean equals(Object o) {
        if (this.getClass() != o.getClass()) return false;
        ConcatenateFunction other = (ConcatenateFunction) o;
        return this.valueSource1.equals(other.valueSource1)
            && this.valueSource2.equals(other.valueSource2)
            && this.delimiter == other.delimiter;
    }
    @Override
    public int hashCode() {
        long combinedHashes;
        combinedHashes = (this.valueSource1.hashCode()
            + this.valueSource2.hashCode())

```

```

        + this.delimiter.hashCode());
        return (int) (combinedHashes ^ (combinedHashes >>> 32));
    }
    @Override
    public String description() {
        return "Concatenates two values together with an optional delimiter";
    }
}

```

关于 `ConcatenateFunction` 类有两个关键点：输入参数和 `getValues` 方法的返回值。`ConcatenateFunction` 类的输入参数由两个 `ValueSource` 对象表示，`ConcatenateFunction` 类会将两个 `ValueSource` 对象包含的值使用连接符 `c` 进行拼接。回顾我们之前讲解的知识，一个 `Function` 的输入参数可以是另一个函数的返回值，通过定义两个 `ValueSource` 对象而不是定义两个 `String` 字符串，你的函数可以接收任意输入。尽管将输入参数统统采用 `ValueSource` 类型来定义带来了很大的灵活性，但 `ConcatenateFunction` 类的构造函数的第 3 个参数 `delimiter` 是一个 `String` 类型，它也可以定义为 `ValueSource` 类型，它还可以是从其他域或者其他函数计算后返回的值，在我们这里，我们假设我们的 `delimiter` 参数是显式的在请求中传递的。为了能够理解 `ConcatenateFunction` 类的输出，你需要查看 `getValues` 方法，这个方法返回一个 `FunctionValues` 对象，并且 `getValues` 方法必须返回 `FunctionValues` 类型，因为我们的 `concatenation` 函数的返回值是一个字符串，我们在内部使用 `StrDocValues` 类表包含这个返回值。`StrDocValues` 类是 `FunctionValues` 类的一个实现类，它能够将 `Integer`、`Boolean` 等类型数据返回成一个 `String`。`FunctionValues` 类有很多子类实现，有些实现可能会使用到特定的缓存，因此如果你需要这方面的优化，那么你需要检出 Solr 的源码进行验证确认。`StrDocValues` 对象内部包含了一个 `strVal(docid)` 方法，当 `Function` 被执行时，它会针对每个索引文档调用一次，正因为如此，所以对于一些执行开销比较大的复杂查询，你需要确保 `strVal` 方法能够尽快执行。

现在你已经知道了 `Function` 是如何计算并返回计算值的，下一步就是理解请求参数是如何传入 `ConcatenateFunction` 对象的。以下代码演示了如何解析输入参数并传入我们的自定义 `Function`：

```

public class ConcatenateFunctionParser extends ValueSourceParser {
    public ValueSource parse(FunctionQParser parser) throws SyntaxError {
        ValueSource value1 = parser.parseValueSource();
        ValueSource value2 = parser.parseValueSource();
        String delimiter = null;
        if (parser.hasMoreArguments()) {
            delimiter = parser.parseArg();
        }
        return new ConcatenateFunction(value1, value2, delimiter);
    }
}

```

以上代码演示了如何使用 `FunctionQParser` 对象将输入参数进行解析并传入我们自定义的函数中。`FunctionQParser` 按照标准的函数语法进行解析，比如 `functionName(input1,input2,...)`，

根据请求的函数名称查找合适的 `ValueSourceParser` 实现类。可以通过调用 `FunctionQParser` 内部的 `parseValueSource()`、`parseArg()`、`parseFloat()` 等方法来获取传递给我们 `Function` 的输入参数。在 `ConcatenateFunctionParser` 示例中，我们期望获取两个 `ValueSource` 对象（可以是一个域，可以是用户输入的任意字符串或者其他函数的返回值）以及一个 `delimiter` 字符串参数，在从请求中读取到这些输入参数之后，我们创建了一个 `ConcatenateFunction` 对象并传入输入参数到其构造函数中。

实现自定义 `concatenation` 函数需要把创建的类都完成，剩下就是在 `solrconfig.xml` 中的 `<<config>>` 元素下进行注册，让 Solr 知道我们自定义的新函数。

```
<valueSourceParser name="concat" class="sia.ch15.ConcatenateFunctionParser" />
```

上面的 `name` 属性表示注册的函数名称，函数名称怎么定义完全由我们决定。`class` 表示我们自定义的 `FunctionParser` 实现类完整包路径。下面是我们自定义的 `concat` 函数的几个使用示例：

```
concat("hello","world", "-")    // 返回 "hello-world"
concat("hello","world", ",")    // 返回 "hello,world"
concat(123,456, ".")             // 返回 "123.456"
concat("no", "delimiter")        // 返回 "nodelimiter"
concat("hello","world", "field1") // 返回 "hellofield1world"
```

如果想要在查询中使用我们刚刚自定义的 `concat` 函数，那么可以这样使用：

```
http://localhost:8080/solr/yourcore/select?q=*&
fl=res:concat(concat(field1,field2,""),"!")
```

上面的查询中我们演示了如何使用 `concat` 函数，先将 `field1` 和 `field2` 这两个域的域值使用逗号连接起来，然后将拼接后的值继续与感叹号！进行拼接。最终 `concat` 函数返回值作为“伪域”以别名 `res` 的方式返回。

11.2 Solr 地理空间查询

Geospatial search 是 Solr 中比较流行的一个功能，它支持基于地理位置进行搜索。实现在每个索引文档中添加一个域，该域一个地理位置坐标点即经度和纬度，然后在查询时请求 Solr 过滤出落在指定坐标点的半径范围内的索引文档。Solr 支持两种主要的地理空间搜索实现方式，旧的实现方式是简单的支持基于单个（纬度，经度）的半径范围内搜索，并且按照距离进行排序。新的地理空间搜索实现方式更复杂一些，它不仅能针对单个坐标点进行过滤，还支持对形状（支持任意复杂的多边形）进行过滤。简单实现主要是在查询时计算两个坐标点的距离，过滤掉相距较远的值。而目前的高级实现是以一系列网格坐标盒子模型来索引形状，搜索转变成跨多个被索引网格的查询，而不需要在查询时计算两点之间的距离。简单实现可以在数据量较小时运行，并且不需要索引额外的数据，然而高级实现方式通常适用

于大数据量的情况下，而且它非常灵活，支持多种形状的搜索。

11.2.1 Solr 地理空间简单查询

Solr 地理空间查询的简单实现支持基于单个坐标点（通常是提供经纬度）进行查询，它能够通过简单的语法来实现圆或正方形范围过滤。

要实现半径范围内搜索，首先你需要在 `schema.xml` 中定义一个域，如下所示：

```
<fieldType name="location" class="solr.LatLonType" subFieldSuffix="_coordinate" />
```

`LatLonType` 类用于地理位置域定义，它包含一对（纬度，经度）坐标值，最后分割两个坐标值映射到单独的域。为了能够成功映射到单独的域，这些域也需要在 `schema.xml` 中存在。通过添加一个以指定后缀结尾的动态域来实现这些单独域的定义，且动态域的域名称后缀需要与域类型的 `subFieldSuffix` 属性值保持一致，以下是配置示例：

```
<dynamicField name="*_coordinate" type="tdouble" indexed="true"
stored="false" />
```

当测试数据导入完成之后，你可以采用多种方式进行位置查询，可以这样查询 `city:"San Francisco, CA"`。即按照城市来查询，但是这种方式并不能查询离你附近位置的索引文档，甚至潜在的还可能会返回不在你查询的城市范围内的文档。

Solr 提供了一种特定的 `Query Parser`（即 `geofilt`），它能解析（纬度，经度）并生成两个单独域同时计算与给定坐标点之间的距离（单位：千米），最后匹配距离指定坐标点指定地理区域内的所有索引文档。比如我们想要查询离 "San Francisco, CA" 半径 20 千米范围内的索引文档，那么查询语法如下所示：

```
http://localhost:8080/solr/geospatial/select?q=*&
fq={!geofilt sfield=location pt=37.775,-122.419 d=20}
```

`sfield` 参数用于指定表示地理位置的域，即在 `schema.xml` 中定义的 `LatLonType` 域，`pt`（即 `point` 的缩写）参数表示纬度、经度的坐标点，`d`（即 `distance` 的缩写）参数用于多少半径范围内（单位：千米），指定为 20 即表示以 20 千米为半径画圆，落在圆内的索引文档将会被返回，如图 11-1 所示。

除了可以实现在指定半径范围内的位置进行查询，还可以通过 `bbox filter` 来实现正方形范围内的位置过滤，`bbox filter` 和 `geofilt` 有点类似，而 `bbox filter` 是根据给定的半径画圆，然后再确定该圆的外切正方形，如图 11-2 所示，`bbox filter` 需要传递的参数与 `geofilt` 一样，但是 `bbox filter` 是以正方形为边界进行过滤的，而确定正方形边界比确定圆计算速度要快，因此通常当要求范围不需要太精确时可以采用正方形边界来过滤，同时性能上稍微比 `geofilt` 好一些。如果你期望是在圆内过滤，但是又希望计算速度快，此时可以使用 `RPT` 域并且尝试调大 `distErrPct` 值比，如 0.1。示例如下：

```
http://localhost:8080/solr/geospatial/select?q=*&
fq={!bbox sfield=location pt=37.775,-122.419 d=20}
```

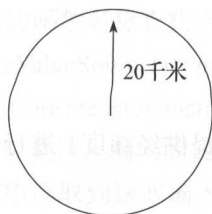



图 11-1 指定半径的圆内查询

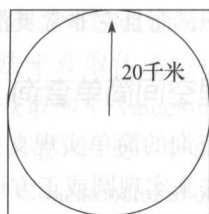


图 11-2 指定半径的 2 倍作为边长的正方形边界内查询

目前为止，你已经知道如何执行一个快速的 `bbox` filter 过滤器去查找地理位置上附近的位置，以及如何使用更精确的 `geofilt` filter 过滤器来过滤指定半径范围内的文档。对于 `geofilt` 而言，Solr 计算了每个索引文档离查询时指定的坐标点之间的距离，因此 Solr 可以在其他方面使用这些距离值。比如 Solr Cloud 允许返回的索引文档能够按照距离值进行排序，或者将距离值随着索引文档一起返回。然而 Solr 已经支持这两个功能。

除了可以根据指定的坐标点对距离值进行过滤之外，Solr 还支持将通过 `geodist` 函数计算得到的距离值作为一个伪域随着查询结果集中的每个索引文档一并返回。之前你已经知道了如何将一个函数的计算值作为一个伪域进行返回，返回 `geodist` 函数的计算值也与之如出一辙。`geodist` 函数的使用语法为：`geodist(sfield, latitude, longitude)`。通过使用 `geodist` 函数，你可以返回每个文档距离指定坐标点的距离值。下面的这个查询示例演示了计算索引文档距离（37.774 93, -122.419 42）这个坐标点的距离值并将其作为伪域返回：

```
http://localhost:8080/solr/geospatial/select?q=*:*&
fl=id,city,distance:geodist(location,37.77493, -122.41942)
```

你会发现返回的结果集中每个索引文档都包含了一个 "distance" 的伪域，同其他 `Function Query` 类似，可以随意返回距离计算函数的返回值作为索引文档的一个伪域一并返回。注意：因为地理位置相关的函数计算都需要执行一些重要的数学计算，而这些数学计算的执行开销往往都很大，因此当你的索引文档达到上百万以上，对每个索引文档执行距离计算函数就会显得很慢，此时应使用 `geofilt` filter 来过滤掉一些索引文档，减少计算量。

你不仅可以将距离计算值作为伪域进行返回，还可以根据距离值对索引文档进行排序。在这点上，`geodist` 函数的使用方式与其他函数并没有什么不同。按照距离从近至远对索引文档进行排序，可以为 `sort` 参数添加 `geodist` 函数，使用示例如下所示：

```
http://localhost:8080/solr/geospatial/select?q=*:*&
fl=id,city,distance:geodist(location,37.77493, -122.41942)&
sort=geodist(location,37.77493, -122.41942) asc,score desc
```

返回的查询结果集部分如下所示：

```
"response":{"numFound":4,"start":0,"docs":[
  {
    "id":"3",
    "city":"San Francisco, CA",
```

```

      "distance":0.03772596784117343}},
    {
      "id":"4",
      "city":"Palo Alto, CA",
      "distance":43.17493506307893},
    {
      "id":"1",
      "city":"Atlanta, GA",
      "distance":3436.669993915123},
    {
      "id":"2",
      "city":"New York, NY",
      "distance":4128.9603389283575]}

```

以上示例中返回的结果集首先按照距离值从小到大进行排序，距离值相同的再按照索引文档评分从高到低进行排序。当然也可以先按照评分排序再按照距离值排序。不仅仅可以结合多个 Function Query，还可以结合相关性评分和地理距离作为复合性相关性评分的一个独立因子。

以上的每个示例都是独立演示的，如果想要在单个查询请求中综合使用距离值过滤、返回距离值为伪域、根据距离值进行排序，那么请看下面这个查询示例：

```

http://localhost:8080/solr/geospatial/select?q=*&
fq={!geofilt sfield=location pt=37.775,-122.419 d=20}&
fl=*,distance:geodist(location, 37.775,-122.419)&
sort=geodist(location, 37.775,-122.419) asc, score desc

```

上面的查询示例看起来有点啰嗦，我们的坐标点参数指定了 3 次，geodist 函数也使用了 2 次。其实你可以将这些输入参数定义在请求的查询文本中，然后 geofilt、bbox 以及 geodist 可以间接地引用这些参数，因此，可以将前面的查询示例进行简化，如下所示：

```

http://localhost:8080/solr/geospatial/select?q=*&
fq={!geofilt}&
fl=*,distance:geodist()&
sort=geodist() asc, score desc&
sfield=location&pt=37.775,-122.419&d=20

```

使用这种简化语法，你还可以覆盖任意地理空间查询组件中显式定义的本地参数，比如 sfield、pt、d 等参数。但是对于一般使用场景来说，这没有必要，因为对于根据距离值过滤、根据距离值排序、返回距离值作为伪域这些操作都是基于相同的坐标点。

11.2.2 Solr 地理空间高级查询

上一节我们学习了 Solr 地理空间查询中简单的基于单个坐标点的文档过滤。Solr 支持更高级的实现，为每个索引文档索引多个坐标点。这种高级地理空间查询支持对任意多边形进行索引而不仅仅是索引单个坐标点。这种功能有什么用途呢？当你想要为每个索引文档索引多个坐标点时会很有用。想象下，假如你的索引数据表示的是一些饭店，其中有些饭店可

能在全国有多个连锁分店, 因此可能有某些饭店对应了多个坐标点, 此时你希望为每个饭店索引多个(纬度, 经度)这样的坐标点, 在查询时, 与索引文档的多个坐标点进行交互。当想要支持任意形状(比如圆、正方形、或者任意的多边形)的区域范围内过滤时, 会使用多边形来描述多个坐标点, 并为每个索引文档索引一个形状。当然也可以采用穷举的方式将任意形状内的多个坐标点索引到索引文档中, 但是当坐标点太多的时候就不太合适了, 此时使用形状去描述更为合理。采用这种方式的话, 当用户执行一个查询, 比如查询 100 千米内的饭店, 此时以 10 千米为半径画圆, 然后每个索引文档对应的多边形与该圆的重叠部分的任意坐标点就是符合要求的。

为了高效的索引任意多边形以及基于任意多边形进行查询, Solr 提供了一个特殊的域类型: `SpatialRecursivePrefixTreeFieldType`。此域类型能够将整个地球划分为一个网格, 在大多数的缩放级别上, 地球都可以划分为带有一定数量区域的网格。这里我们将使用四个象限为例进行类比说明, 每个象限又会被划分为更小的 4 个象限, 你可以递归任意深度继续划分下去, 图 11-3 形象展示了如何将象限划分为 3 层。

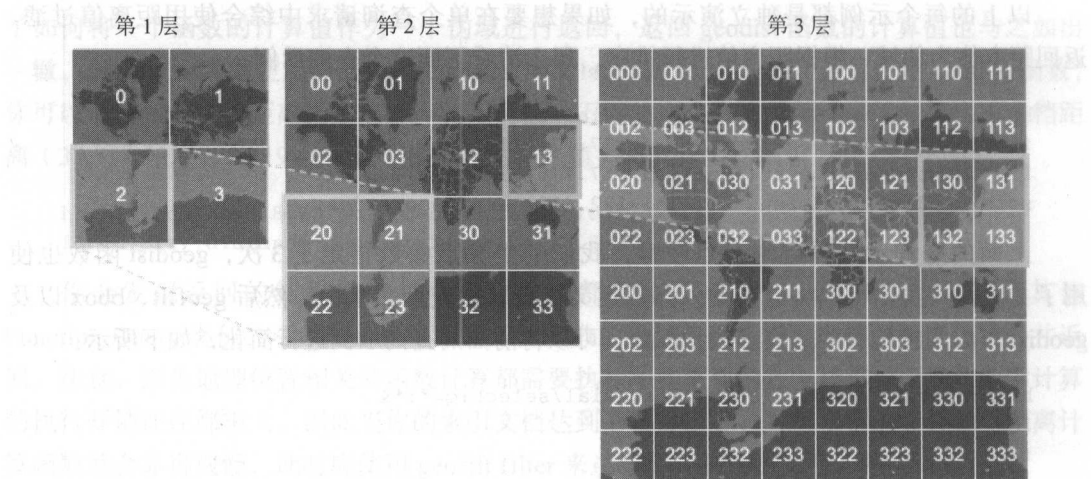


图 11-3 将整个地球划分为 3 个层级的象限, 每一层可以继续划分 4 个象限

从图 11-3 中可以看出, 网格中的每个盒子都包含了一个唯一标识符来表示它所在层级和精度的等级。通过这种方式对整个地球进行建模, 这样就能高效地基于位置进行查询而不是计算距离值。比如图 11-3 中, 假如你要定位某个城市, 首先会在第 1 层中定位到 box 0, 然后会定位到 box 03, 最后会定位到 box 032 (当然可以划分更多层以提高查询精度), 然后我们可以在 Solr 索引中对 3 个层进行索引, 这样使得地理空间查询变得非常强大。假如想要查询西半球的索引文档, 此时你只需要根据 Term: "0" OR "2" 在索引中的位置域上进行查询即可, 因为西半球在图 11-3 的 0 和 2 区域。对网格进行索引然后基于网格进行查询, 查询速度通常会比计算每个索引文档离坐标点的距离值要快得多。

尽管从技术层面上来讲,你没必要去理解:Solr 地理空间查询底层的基于网格的分层思想将索引位置转变成索引网格的 gridID (网格的唯一标识符),根据 gridID 能知道当前位置处在网格中哪一层哪个格子中。但是有个基本的了解有助于你更好地使用 Solr 的地理空间查询,比如如何在索引创建或查询时根据你的地理位置要求索引的精度来选择一些配置项。

在 Solr 中,针对上述原理提供了不同的索引方式去实现,分别是 GeohashPrefixTree 类和 QuadPrefixTree 类,其中 GeohashPrefixTree 对应 GeoHash 算法(也采用了笛卡尔层的分层思想),QuadPrefixTree 对应笛卡尔层的实现,两者都是基于 PrefixTree 算法实现。

基于网格的位置系统利用前缀树的概念将位置表示为一系列的前缀字符(比如 31102)。SpatialRecursivePrefixTreeFieldType 提供了一个前缀树的抽象表示,它允许使用多网格分层的方式来索引和查询地理空间数据,上面的示例描述了一个简单的 4 网格实现。另外一个前缀树实现是基于著名的 Geohash 算法,它几乎是行业内的标准,因此它也是 Solr 中默认的实现。Geohash 是专门设计用于对地球的地理位置进行建模的。当你在 schema.xml 中将 SpatialRecursivePrefixTreeFieldType 的 geo 属性设置为 true,那么 Geohash 实现就会被启用,其实默认 geo 属性就是 true。Geohash 标准文档很齐全,理解 Geohash 算法也不是本章的重点,因为它在概念上与 4 网格的前缀树实现基本相似,想要更详细地学习了解 Geohash,请搜索 "Geohash 算法原理",会找到很多文章讲解 Geohash 算法。

GeohashPrefixTree 与 QuadPrefixTree 都继承自 SpatialPrefixTree 这个前缀树基类,都使用了网格分层思想,主要区别就是索引和查询逻辑不一样,比如,默认网格分层的层级不同,获取子网格的方式不一样。GeohashPrefixTree 有 32 个子网格(编码为 0-z),QuadPrefixTree 只有 4 个子网格,编码为 ABCD,其中 A 为左上, B 为右上, C 为左下, D 为右下,即相当于 4 个象限。

基本理解基于网格的搜索系统能够让你充分理解如何高效地使用 Solr 中的地理空间高级查询,这一切关于将复杂的地理坐标和区域形状映射成网格坐标系全部都封装在 SpatialRecursivePrefixTreeFieldType 类内部,你需要做的是在 schema.xml 中定义这个域类型,示例如下所示:

```
<fieldType name="location_rpt" class="solr.SpatialRecursivePrefixTreeFieldType"
  spatialContextFactory=
    "com.spatial4j.core.context.jts.JtsSpatialContextFactory"
  distErrPct="0.025" maxDistErr="0.000009"
  autoIndex="true" distanceUnits="degrees" />
<field name="location_rpt" type="location_rpt" indexed="true"
  stored="true" multiValued="true" />
```

其中:

□ JtsSpatialContextFactory: 当有 Polygon 多边形时会使用 jts (需要把 jts.jar 放到 Solr war 包的 lib 目录下)。基本形状使用 SpatialContext (spatial4j 的类)。

❑ **distErrPct** : 定义非 Point (坐标点) 形状的精度, 范围在 0 ~ 0.5 之间, 默认 0.025。该值决定了非 Point 的形状在索引或查询时的 level (层级) (如 geohash 模式时就是 geohash 的长度)。distErrPct 设置为 0 时取 maxLevels (最大层级), 即精度最大。

❑ **units** : 表示计算单位, 如数学中的角度 degrees。此参数已经过时了, 不推荐使用, 建议使用 distanceUnits 参数代替 units, 参数可选值有 degrees (角度), kilometers (千米), miles (英里)。

❑ **format**: 表示形状定义所使用的语法, 可选值有 WTK (默认值)、GeoJSON、LEGACY、POLY, 关于其他语法请访问如下链接详细了解:

[https:// locationtech.github.io/spatial4j/apidocs/org/locationtech/spatial4j/io/PolyshapeReader.html](https://locationtech.github.io/spatial4j/apidocs/org/locationtech/spatial4j/io/PolyshapeReader.html)。

❑ **maxDistErr** : 定义非 Point (坐标点) 形状的最大精度, 如果此参数未指定, 默认值是 1 米即稍微比 0.000 009 度小一点点, 此参数设置主要用于内部决定合适的最大层级。

❑ **worldBounds** : 定义在 ENVELOPE(minX, maxX, maxY, minY) 语法中, x 和 y 的合法数字范围, 如果域类型的 geo="true", 就使用标准的 lat-lon (纬度-经度) 来定义整个地球边界, 如果域类型的 geo="false", 你需要定义你自己的边界。

❑ **distCalculator** : 定义距离计算使用的算法, 如果域类型的 geo="true", 那么默认使用 "haversine" 算法; 如果域类型的 geo="false", 则默认会使用 "cartesian" 算法, 其他可选值有 "lawOfCosines"、"vincentySphere"、"cartesian^2"。

❑ **prefixTree**: 定义网格的具体实现, 因为 PrefixTree (前缀树, 比如 RecursivePrefixTree) 将整个地球映射成一个网格, 网格中的每个格子又会继续划分成下一个层级的网格。如果域类型的 geo="true", 那么默认的前缀树实现就是 geohash, 否则就是 quad (四网格形式), Geohash 的每个层级采用 32 个格子, 而 quad 每个层级只有 4 个格子。第 3 种选择就是 packedQuad, 通常它比纯 quad 的四网格形式性能更高效, packedQuad 提供了大约 20 个层级, 你可以设置更多。

❑ **maxLevels** : 设置索引数据时网格层级的最大深度, 一般通过指定 maxDistErr 参数来控制最终计算得到一个合适的最大层级深度数值会更直观一些。

❑ **geo** : 如果将 geo 参数设置为 true, 那么会使用纬度经度坐标, 此时的数学模型通常是球体; 如果将 geo 参数设置为 false, 那么坐标点会是二维空间上的欧几里得几何或者笛卡尔几何中常见的 (x, y) 形式。

坐标点可以采用传统的逗号分割方式来定义, 比如纬度、经度, 或者忽略逗号直接使用一个空格字符分隔, 比如纬度经度, 也可以采用 POINT(x, y) 语法来定义, 示例如下所示:

```
<field name="location_rpt">43.17614,-80.57341</field>
<field name="location_rpt">-80.57341 43.17614</field>
<field name="location_rpt">POINT(-80.57341 43.17614)</field>
```

你可以使用 LINESTRING 语法来表示线段, 示例如下所示:

```
<field name="location_rpt">LINESTRING(0 0,1 0,0 2)</field>
```

一个长方形有 4 个顶点，因此需要用 4 个点用一个空格字符分别分割来表示，4 个点依次表示顺序为 MinX、MinY、MaxX、MaxY。示例如下所示：


```
<field name="location_rpt">-74.093 41.042 -69.347 44.558</field>
```

上述代码表示一个圆需要圆心坐标和一个半径，为了表示更复杂的形状，Solr 提供了特定的语法来表示这些形状，你需要将输入参数包裹在特定语法内，示例如下所示：

```
<field name="location_rpt">Circle(37.775,-122.419 d=20)</field>
```

圆心坐标你依然可以采用前面的坐标点表示语法来进行定义，圆的半径通过 d 参数来指定（单位：千米）。

为了能够支持对任意复杂的形状进行表示，Solr 提供了使用 WKT (well-known text) 标准来定义任意的多边形。尽管在 Solr 中常用的形状一般是坐标点、圆、长方形，Solr 通过使用 Apache 的 Spatial4J 类库来对这些形状进行支持，通过使用 WKT 来支持对任意多边形的表示，使用 WKT 需要额外添加 JTS(Java Topology Suite 的缩写，即 Java 拓扑套件) 依赖。

 **注意** Apache Solr 是基于 Apache 2.0 License 开源的，它允许你在自己的系统中使用 Solr 的任意代码，你没有义务必须分享你的代码或者支付任何许可费用。JTS 使用的是 LGPL 协议，JTS jar 包并没有包含在 Solr 源码中。如果你想要在 Solr 中使用 JTS，那么你需要额外添加 JTS jar 包依赖，只要你不修改 JTS 的源码，那么你引用 JTS 通常认为是安全的，但是如果你修改了 JTS 源码，那么你就必须要开源你的项目。因此当你选择考虑使用 JTS 时，你需要考虑这个问题。

想在 Solr 中启用 WKT 支持，需要添加 jts jar 包依赖到 solr.war 包的 WEB-INF/lib 目录下，即 apache-tomcat-7.0.55\webapps\solr\WEB-INF\lib 下。jts jar 包随书源码中有提供，也可以通过网络搜索下载获取到。请从随书源码中获取 "geospatial_jts" Core 的配置文件创建 Core，并通过执行 IndexGeospatialJTS 类导入测试数据，然后重启 Solr Server，这样你的 JTS 就正式启用了，我们可以演示一些强大的地理空间形状索引和查询功能。WKT 允许你使用如下的语法来定义任意的多边形：

```
<field name="location_rpt">
POLYGON((-10 30, -40 40, -10 -20, 40 20, 0 0, -10 30))
</field>
```

使用 JTS 的语法来定义任意多边形时你需要记住以下几点：

- 每个坐标点之间使用逗号分隔；
- 每个坐标点定义格式为“经度纬度”；
- 如果你想要定义闭合的形状，那么第一个坐标点和最后一个坐标点定义需要相同；

❑ WKT 坐标点会映射到有效的经度 ($\pm 90^\circ$) 和纬度 ($\pm 180^\circ$) 范围内;

❑ 只有圆支持南北极。

通过启用对任意多边形的索引, 你就可以在 Solr 中基于这些形状进行搜索, Solr 的强大地理空间查询功能也随之解锁。请执行下面这 2 个查询示例, 如下所示:

```
http://localhost:8080/solr/geospatial_jts/select?q=*&
fq={!geofilt pt=37.775,-122.419 sfield=location_rpt d=5}
http://localhost:8080/solr/geospatial_jts/select?q=*&
fq={!bbox pt=37.775,-122.419 sfield=location_rpt d=5}
```

我们除了能够使用类似上面示例那样基于圆和正方形来查询, 还可以使用相同的形状, 使用 `SpatialRecursivePrefixTreeFieldType` 提供的更高级的基于网格的前缀树系统来执行查询, 这类查询的语法如下所示:

```
http://localhost:8080/solr/geospatial_jts/select?q=*&
fq=location_rpt:"Intersects(POLYGON((-10 30, -40 40, -10 -20, 40 20, 0 0, -10
30)))"
```

上面的查询示例中我们定义了一个五边形, 然后 `Intersects` 表示相交, 即索引文档中我们在 `location_rpt` 域上索引的多边形或者圆等形状与查询时指定的形状时相交重叠, 如果两者有相交, 那么就返回索引文档。

除了可以使用 `Intersects` (相交) 操作, 针对 `SpatialRecursivePrefixTreeFieldType` 域进行查询, 还支持 `IsWithin`、`Contains`、`IsDisjointTo` 操作, 具体如表 11-6 所示。

表 11-6 四类匹配操作

操 作	描 述
<code>Intersects</code>	匹配两个形状相交部分的索引文档
<code>IsWithin</code>	匹配被包含在查询时指定的形状内部的索引文档
<code>Contains</code>	匹配包含查询时指定的形状的索引文档
<code>IsDisjointTo</code>	匹配包含两个形状互斥不相交部分的索引文档, 前提是两个形状要相交

使用上面表格中的操作, 你可以查询被包含在指定的多边形内部的所有索引文档, 查询示例如下所示:

```
http://localhost:8080/solr/geospatial_jts/select?q=*&
fl=id,location_rpt,city&
fq=location_rpt:"IsWithin(
POLYGON(
-85.4997 34.7442,
-84.9723 30.6134,
-81.2809 30.5255,
-80.9294 32.0196,
-83.3024 34.8321,
-85.4997 34.7442))
) distErrPct=0"
```

使用 Boolean 操作符 (AND 和 OR) 来连接多个形状查询, 从而构建一个复杂的查询。你不仅可以指定形状进行查询, 还可以使用两个形状之间的距离来影响你的查询相关性, 后续章节会详细讲解如何使用地理空间距离值来干预你的 q 参数构造的主查询的相关性评分。

除了 SpatialRecursivePrefixTreeFieldType, 还可以使用 RptWithGeometrySpatialField, 它是 SpatialRecursivePrefixTreeFieldType 的派生子类, 会额外在 DocValues 中存储原始几何结构, 使用它能够进行精确查询。它同时还支持对坐标点进行索引。它的相交 (默认操作) 判定操作速度非常快, 因为很多搜索结果可以作为一个精确命中而不需要返回进行几何检查。这个域类型与 SpatialRecursivePrefixTreeFieldType 域类型配置没什么太大不同, 它的 distErrPct 参数默认值是 0.15, 比 SpatialRecursivePrefixTreeFieldType 默认的 0.025 要大, 因为使用网格纯粹是出于性能考虑, 而不是真正去表示形状。

当索引数据中索引的形状包含了很多顶点时, 你可以配置使用如下这个缓存, 比如有个域叫 "geom", 可以在 solrconfig.xml 中配置一个可选的缓存, 配置如下所示:

```
<cache name="perSegSpatialFieldCache_geom"
      class="solr.LRUCache"
      size="256"
      initialSize="0"
      autowarmCount="100%"
      regenerator="solr.NoOpRegenerator"/>
```

当使用 RptWithGeometrySpatialField 这种域类型时, 你可能不希望将这个域设置为 stored="true", 因为它已经存储了冗余的 DocValues 值。但是你又想在查询返回的结果集中返回这个域的域值, 此时你可以使用 geo formatter 来获取:

```
f1=geojson:[geo f=mySpatialField w=GeoJSON]
```

这里的 f 表示你的域名称; w 表示形状定义语法格式, 可选值为 WTK 和 GeoJSON; 前面的 geojson 是返回的伪域的别名。

此外还可以使用 BBoxField 域来对长方形进行索引, 并且 BBoxField 域支持对边界框进行查询, 它支持大部分的地理空间查询判断 (比如相交、包含), 而且还为重叠部分或者两个长方形中间区域提供了增强型相关性模型。特别是它的相关性模型会非常有用, 在 schema.xml 中配置 BBoxField 域的示例如下所示:

```
<field name="bbox" type="bbox" />
<fieldType name="bbox" class="solr.BBoxField"
  geo="true" units="kilometers" numberType="_bbox_coord" storeSubFields="false"/>
<fieldType name="_bbox_coord" class="solr.TrieDoubleField" precisionStep="8"
  docValues="true" stored="false"/>
```

BBoxField 实际上是基于 4 个其他数字域类型实例, 它使用一个 boolean 变量标识国际日期变更线交叉。如果你想要使用它的相关性功能, 那么 docValues 必须设置为 true。对于

geo、worldBounds、spatialContextFactory 等参数域 RPT（即 SpatialRecursivePrefixTreeField Type 域类型的简称）域类型，创建 BBox Field 会有所不同。比如你想要索引一个长方形，此时你需要添加一个 bbox 域，使用 WKT/CQL 的 ENVELOPE 语法来定义形状的数据：

```
<field name="location_rpt">ENVELOPE(-10, 20, 15, 10)</field>
```

4 个参数依次表示 minX、maxX、maxY、minY。你可以使用使用 WTK 的 POLYGON 语法来定义一个矩形，或者使用 GeoJSON 语法（当设置 format="GeoJSON"，GeoJSON 语法会被启用）。

查询的时候，可以使用 {!bbox} 查询解析器，示例如下所示：

```
&q=*&fq={!bbox sfield=yoursfield}&pt=45.15,-93.85&d=5
或者
&q={!field f=bbox}Contains(ENVELOPE(-10, 20, 15, 10))
```

还可以通过 score 参数来指定相关性模型，示例如下所示：

```
&q={!field f=bbox score=overlapRatio}Intersects(ENVELOPE(-10, 20, 15, 10))
```

score 参数可选值有 overlapRatio、area、area2D 等。

area 通过球面数学（假定 geo=true）进行打分，area2D 简单使用 width * height，overlapRatio 基于索引文档的形状重叠的区域面积大小来进行打分（分值范围为 [0-1]）。关于 overlapRatio 的计算公式更详细的内容请查阅 BBoxOverlapRatioValueSource 类的源码。此外还有一个额外的 queryTargetProportion 参数，它允许划分查询端和索引段权重比例值，取值范围为 [0-1]，0.5 表示两边权重相同。你可以在查询时通过添加 &debug=results 参数来查询更多有关评分计算的详细信息。

geofilt 和 bbox 支持一些共同参数，如表 11-7 所示。

表 11-7 geofilt 和 bbox 支持的参数

参 数	描 述
d	表示半径，单位默认是千米，对于 RPT & BBoxField 域，你可以通过 distanceUnits 参数设置为其他单位
pt	表示中心坐标点，如果 geo=true，那么就是 "纬度，经度" 这种格式，如果 geo=false，对于 RPT 域，那就是 "x y" 这种格式，否则就是 "x, y" 这种格式
sfield	表示地理空间索引域的域名称
score	这是一个高级配置项，仅仅针对于 RPT 和 BBoxField 域。如果查询是在 q 参数构造的主查询上下文环境下，那么会计算文档评分，可选值如下： <input type="checkbox"/> none：不计算文档评分，所有文档固定评分为 1.0（默认值）。 <input type="checkbox"/> kilometers：计算每个文档的域值跟指定的中心坐标点之间的距离值，单位为千米。 <input type="checkbox"/> miles：同上，只是单位为英里。 <input type="checkbox"/> degrees：同上，只是单位为度数（经纬度）。 <input type="checkbox"/> distance：计算每个文档的域值跟指定的中心坐标点之间的距离值，单位由 distanceUnits 参数决定。

(续)

参 数	描 述
score	<input type="checkbox"/> recipDistance: 同 distance 类似, 但是 recipDistance 是 distance 的倒数。 <input type="checkbox"/> BBoxField 域的 score 参数还额外支持如下可选值: <input type="checkbox"/> overlapRatio: 基于索引文档的形状重叠的区域面积大小来进行打分 (分值范围为 [0-1])。 <input type="checkbox"/> area: 通过球面数学里的 haversine 函数 (此时要求 geo=true) 对两者形状重叠部分进行打分, 返回值以 distanceUnits 参数配置的单位形式返回。 <input type="checkbox"/> area2D: 采用二维空间中的笛卡尔几何来对两者形状重叠部分进行打分, 返回值以 distanceUnits 参数配置的单位形式返回
filter	这是一个高级配置项, 仅对 RPT 和 BBoxField 域设置有效。如果你想要在查询时对索引文档进行评分操作 (评分方式由上面的 score 参数决定) 且此查询不是 Filter Query, 那么请将此参数设置为 false

我们在函数查询章节中介绍过 geodist() 这个地理空间计算函数, 它同样适用于 RPT 域 (除了适用于 LatLonType 域之外)。然而遗憾的是, 当使用前缀树实现来索引形状时会损失一些精度, 因此此时通过 geodist() 函数计算的距离值可能会不太精确, 但是这并不是什么大问题, 只要你将 distErrPct 参数设置的足够高。你可以就通过 geofilt 操作返回每个索引文档的距离值来模拟 geodist() 函数。

回顾我们学习的 geofilt query parser, 它包含了一个性能优化点。首先 geofilt 会使用形状的边界过滤掉一些索引文档, 然后计算边界内剩下的坐标点离中心坐标点的距离。因为 geofilt 会计算每个索引文档和查询时指定的坐标点之间的距离, 因此你可以将这个距离值作为评分随着索引文档一并返回, 具体请看下面的查询示例:

```
http://localhost:8080/solr/geospatial_jts/select?sort=score asc&
q={!geofilt pt=37.775,-122.419 sfield=location d=5 score=distance}
```

由于 geofilt 应用于 q 参数的主查询, score=distance, 那么意味着返回的每个索引文档的评分计算方式就变成了 geofilt 计算的距离值了。除了可以指定 score=distance, 你还可以指定 score=recipdistance, recipdistance 是在 geofilt 计算的距离值基础上再求倒数。如果你未指定 score 参数, 那么 score 默认值是 none, 此时返回的每个索引文档的评分都是固定的 1.0。

如果想要为所有索引文档进行评分, 尽管那些索引文档并不在 geofilt 过滤范围内, 此时可以设置 filter=false 来关闭 geofilt query parser 的文档过滤功能, 示例如下所示:

```
http://localhost:8080/solr/geospatial_jts/select?sort=score asc&
q={!geofilt pt=37.775,-122.419 sfield=location_rpt d=5
score=distance filter=false}
```

通过将 filter 关闭, 可以将 geofilt 转变成跟 geodist 函数一样用途, 即都能返回距离值, 同时不再使用 filter 来过滤索引文档, 最后将距离值作为每个索引文档的评分。遗憾的是, 因为 geofilt 并不是一个 Function Query, 所以如果想要根据 geofilt 返回的距离值进行排序同时又想要在函数中使用 geofilt, 那么此时可以如下执行查询:

```
http://localhost:8080/solr/geospatial/select?
sort=$distance asc&
fl=id,distance:$distance&q=*&
distance=query($distFilter)&
distFilter={!geofilt pt=37.775,-122.419 sfield=location_rpt d=5
score=distance filter=true}
```

上面的查询示例通过 `query` 函数将 `geofilt` 查询包装成一个 `Function Query`，通过变量引用函数返回的距离值，然后可以在多个地方应用这个距离值，比如可以将这个变量作为伪域返回，可以根据这个变量表示的距离值对索引文档进行排序。

第 6 章里我们学习了如何使用 `Facet` 对任意查询进行统计。通过结合多个 `geofilt`，可以很容易实现基于地理空间距离进行有趣的数据分析统计。为了使本章的学习更有趣，我们准备索引 10 万条索引文档。导入这些测试数据，请执行随书源码中的 `DistanceFacetDocGenerator` 测试类，导入之前请先创建好 "distancefacet" `Core`。

导入成功后看到打印的提示信息："totle documents: 103864"。每个索引文档包含了一个 `location` 域（用经纬度表示），同时还有 `city` 域。我们可以创建一个 `Facet` 查询，比如统计 10 千米以内、20 千米以内、50 千米以内、100 千米以内等范围内的索引文档。为了展示一个更复杂的查询示例，我们假设执行一个 `Facet` 查询统计方圆 50 千米以内的前 10 位 `city`，然后统计方圆 20 千米以内的前 10 位城市的统计数量。看起来蛮复杂的，下面的查询示例演示了如何实现上面的需求：

```
http://localhost:8080/solr/distancefacet/select?q=*&rows=0&
fq={!geofilt sfield=location pt=37.777,-122.420 d=80}&
facet=true&facet.field=city& facet.limit=10
```

返回的查询结果集如下所示：

```
"facet_fields":{
  "city":[
    "San Francisco, CA",11713,
    "San Jose, CA",3071,
    "Oakland, CA",1482,
    "Palo Alto, CA",1318,
    "Santa Clara, CA",1212,
    "Mountain View, CA",1045,
    "Sunnyvale, CA",1004,
    "Fremont, CA",726,
    "Redwood City, CA",633,
    "Berkeley, CA",599]],
```

先热身，接下来我们要放大招了，请看下面的查询示例：

```
http://localhost:8080/solr/distancefacet/select?q=*&rows=10&
distanceFilter={!geofilt sfield=location pt=37.777,-122.420 score=distance
filter=true d=20}&
facet.limit=10&facet=true&facet.field=city&facet.sort=count&
```

```
distance=query($distanceFilter)&sort=$distance asc&fl=id,city,distance:$distance&
fq=_query_:(!geofilt sfield=location pt=37.777,-122.420 d=20}
```

上面的查询示例中，我们首先 `q` 参数返回所有索引文档，其次执行 `fq` 的 `geofilt` 过滤查询，返回以 `(37.777, -122.420)` 为圆心，20 千米为半径的圆内的索引文档。然后按照 `city` 域对剩余的索引文档进行统计，对 `facet` 统计返回的结果集按照数量从高到低排序，同时限制 `Facet` 统计返回结果集数量为 10，即实现对 Top 10 城市的统计。最后对返回的结果集中的索引文档按照 `distanceFilter` 计算的距离值从小到大进行排序，即由近至远进行排序，最终返回 `id`、`city`、`distance` 这 3 个域，其中 `distance` 表示距离值的伪域。`fq` 查询中我们通过 `_query` 语法将 `geofilt` 过滤转换成了伪 Term 查询，而 `_query` 类似于伪域，看起来就好比在指定域上执行简单的 Term 查询，比如 `title:solr`。你可以定义多个 `_query` 来构造这种伪 Term 查询，也可以使用 AND 或 OR 操作符来连接多个 `_query` 查询表达式，从而构造出一个超复杂的查询。

此刻，你已经学会了 Solr 中的地理空间查询，下一节我们将继续学习 Solr 中的一些更有趣的数据统计分析功能：Pivot Facet。

11.3 Pivot Facet

在第 6 章，我们已经学习了 Solr 中的 Facet 查询，包括 Query Facet（基于一个查询进行 Facet）、Facet Filter（结合 Filter Query 来过滤 Facet）、Range Facet（实现 Facet 区间范围查询统计）、Field Facet（基于域进行 Facet）等。但是有时候，实现嵌套 Facet 会很非常有用，这就好比传统关系型数据库中，我们可以 `group by a,b` 对 `a` 和 `b` 两个字段进行分组统计是类似的。Solr 支持这种嵌套 Facet 来实现类似 SQL 里的同时按多个字段 `group by` 的功能：Pivot Facet。

Pivot Facet 支持多某个域进行 Facet。比如你的关于“饭店”的索引数据中包含了 `state`、`city`、`rating` 这 3 个域，`rating` 域表示饭店的星级，用 1 ~ 5 颗星星表示。当想要统计各个城市各个州下 4 ~ 5 星级酒店总数时，就可以使用 Pivot Facet 功能来实现类似这种需求。

首先还是跟之前的章节一样，请根据随书源码提供的文件创建“pivotfaceting”Core，并执行 `IndexPivotFacet` 测试类导入测试数据到“pivotfaceting”Core 中，为我们接下来的查询测试做好准备。

假定此时你已经成功导入了测试数据，那么请随我一起执行下面这个 Facet 查询示例：

```
http://localhost:8080/solr/pivotfaceting/select?q=*&
fq=rating:[4 TO 5]&facet=true&facet.limit=3&
facet.pivot=mincount=1&facet.pivot=state,city,rating
```

返回的结果集部分如下所示：

```
acet_pivot":{
  "state,city,rating":[{
    "field":"state",
    "value":"GA",
    "count":4,
    "pivot":{
      "field":"city",
      "value":"Atlanta",
      "count":4,
      "pivot":[{
        "field":"rating",
        "value":4,
        "count":2},
        {
          "field":"rating",
          "value":5,
          "count":2}]]}],
```

以上的查询示例演示了 Pivot Facet 的 3 级嵌套，通过 `facet.pivot` 参数定义了 3 个使用逗号分隔的域，不像我们第 6 章里学习的根据单个域进行 Facet 只会单个 Facet 统计值。Pivot Facet 除了会返回当前域上的 Facet 统计值，还会继续按照下一个域对当前域的 Facet 统计值进行分组细分统计，你可以这样一直嵌套下去。这种嵌套 Facet 在实际中很有用，比如当你在搜索“手机”，你首先关心的是手机品牌，如果网站将手机数据按照品牌分组，这样就能快速帮你过滤掉其他不喜欢的品牌的手机数据，紧接着你可能关心的是手机的价格，因此此时可以在原来品牌手机的基础再按照不同价格区间对手机进行分类统计，比如你购买手机的预算是 5000 元左右，那么不符合这个价格区间范围的手机又被过滤掉，下一步你可以会挑选手机颜色，就这样层层叠加多个维度对手机数据进行分类统计在实际项目中很常见也很有用。这跟 SQL 里的 Group by 多个字段的特性有种似曾相识的感觉。

假如有这样一个需求，你想要统计每个班级下 60 ~ 70 分、70 ~ 80 分、80 ~ 90 分、90 ~ 100 分各个分数段之间的人数。这里的分数区间查询可以使用我们前面学习过的 `facet.range` 来实现，统计每个班级显然是根据班级进行分组，这里班级和分数区间相当于是分组的两个维度即 `Facet`，现在要求在班级这个维度再叠加一个分数区间维度进行统计，此时我们需要使用 `Pivot Facet` 来实现，查询示例如下所示：

```
http://localhost:8080/solr/yourcore/select?q=*&
facet=true&facet.range={!tag=grade}fenshu&
f.fenshu.facet.range.start=60&f.fenshu.facet.range.end=100&
f.fenshu.facet.range.gap=10&
facet.pivot={!range=grade}classes
```

这里的 `classes` 表示班级 `Field`, `fenshu` 表示分数 `Field`, 我们先在外面按照分数域定义了 `facet.range` 区间范围查询, 然后通过 `{!range=grade}` 语法在 `facet.pivot` 上叠加了区间范

围这个维度进行统计，这样就实现了我们的需求。同理你还可以使用 facet.query 并将 facet.query 作为一个 2 级维度叠加嵌套到 facet.pivot 所在域即 classes 域上，比如你想要统计每个班级姓“杜”的有多少人，那么你可以这样查询实现，示例如下所示：

```
http://localhost:8080/solr/yourcore/select?q=*&
facet=true&facet.query={!tag=nameFilter}stuName:杜*&
facet.pivot={!query=nameFilter}classes
```

当然你可以将 query 和 range 在 pivot facet 中结合在一起，比如：

```
facet.pivot={!range=grade query=nameFilter}classes
```

但是需要注意的是，上面的 grade 和 nameFilter 是同级 Facet，即总共只有 classes 和 grade/nameFilter 两级 Facet，也就是说它们是分别在 classes 域 Facet 之后的基础上再 Facet 统计。如果你想要实现两个 Facet Query 之间的 Facet 嵌套，即将一个 Facet Query 作为一个维度进行 Facet 统计，然后在其基础之上再定义一个 Facet Query 作为 2 级维度继续 Facet 统计，很遗憾，Solr 暂时还不支持这种任意两个 Query 定义的 Facet 之间的嵌套统计查询。虽然 Pivot Facet 是用来设计解决多维度叠加统计问题的，但是 Pivot Facet 目前只支持先按照指定的域作为第一级维度进行 Facet 统计，然后二级维度可以继续是一个域，也可以是任意 Query。

Pivot Facet 其实还有一定的性能约束。因为 Pivot Facet 需要将一个 Facet 再细分为多个子 Facet，与单独执行每个 Facet 统计相比，为每个 Facet 计算子层级的 Facet 这需要额外的计算执行开销。通常我们最关心的是 Pivot Facet 对大数据量进行统计时性能如何。比如我们的第一个 Pivot Facet 中我们按照 state、city、rating 这 3 个 Facet 进行统计，它们依次表示了 3 个层级的 Facet，一级一级地将 Facet 细分下去，所以我们得到 $3 \times 3 \times 3 = 27$ 个 Facet，如果每个 Facet 下有 100 或 1000 个甚至 10 000 个唯一值的话，那返回的结果集将会非常大。而且 Pivot Facet 返回的结果集数据比普通的 Facet 更冗长，很容易就使得返回的结果集大小达到几 MB，当然你可以通过 facet.limit 参数来限制每个 Facet 内返回的记录条数来控制返回的结果集字节大小。而假如你的需求确实需要每组返回 Top 100 或 Top 1000，那么你的 Java 堆内存可能会撑爆，内存会被瞬间消耗掉，其他查询自然无法充分利用缓存，查询速度也会随之下降，甚至出现响应卡顿。鉴于以上原因，当你考虑需要根据几个域进行嵌套统计或者每个 Facet 下最多返回多少个值时，你需要格外谨慎，心中要清楚可能会导致的后果。

11.4 Solr Subfacet

Solr 中的 Subfacet 又称为 Nested Facet（嵌套 Facet）。它是 Solr Pivot Facet 的升级版，它允许你为父 Facet 生成的每个 bucket 添加额外的子 Facet。Subfacet 相比 Pivot Facet 有以下几点优势：

- ❑ Subfacet 使用 Facet Function (Facet 函数) 来进行统计, 具有强大的实时分析能力;
- ❑ 可以将一个 Subfacet (子 Facet) 嵌套到任意的 Facet 类型 (比如 Field、Query、Range) 中;
- ❑ 一个 Subfacet (子 Facet) 可以是任意的 Facet 类型 (比如 Field、Query、Range);
- ❑ 一个给定的 Facet 可以嵌套多个 Subfacet (子 Facet);
- ❑ 就像顶级 Facet 一样, 每个 Subfacet (子 Facet) 可以拥有它自己单独的配置 (比如 offset、limit、sort、stats 等)。

11.4.1 Subfacet 语法

Subfacet 是新 Facet 功能模块中的一部分, 采用 JSON Facet API 形式进行表示。每个 Facet 命令其实就是一个 Subfacet。因为每个主查询或 Filter Query 匹配的索引文档都会隐式的定义一个顶级的 Facet Bucket, 你可以简单地通过声明 Facet 命令参数来添加一个 Facet, 比如在 “genre” 域上简单的添加一个 “terms” 类型的 Facet, 示例如下所示:

```
top_genres:{
  type: terms,
  field: genre,
  limit: 5
}
```

如果你想要在 genre 这个 Facet 基础上再添加一个 SubFacet 来返回前 4 名作者, 那么可以这样定义:

```
top_genres:{
  type: terms,
  field: genre,
  limit: 5,
  facet:{
    top_authors:{
      type: terms,
      field: author,
      limit: 4
    }
  }
}
```

11.4.2 Subfacet 复杂示例

假设你想要执行类似下面这样的复杂 Facet 查询:

- ❑ 首先你想要按照 genre 域进行 Facet 统计;
- ❑ 在 genre 域 Facet 基础之上再按照 author 域进行 Facet 统计, 每个 author 子 Facet 内返回前 7 名作者;
- ❑ 同时还要在 genre 域 Facet 基础之上创建一个 “highpop” (即高人气的意思) 的 Facet 进行统计 popularity 人气指数在 [8-10] 之间的作者;

□ 最后在每个 “highpop” Facet 基础之上再创建一个 “publisher” Facet 进行统计，返回高人气作者中的前 5 名所属出版社。

简而言之，你想要统计每个书籍分类下前 7 名作者，同时统计每个书籍分类下畅销书（高人气的书即表示该书畅销）中的前 5 名所属出版社。

使用 JSON Facet API 来表示以上的 Facet 查询，可以如下所示：

```
json.facet=
{
  top_genres:{
    type: terms,
    field: genre,
    facet:{
      top_authors: {
        type: terms,
        field: author,
        limit: 7
      },
      highpop:{
        type: query,
        q: "popularity:[8 TO 10]",
        facet:{
          publishers:{
            type: terms,
            field: publisher,
            limit: 5
          }
        }
      }
    }
  }
}
```

Subfacet 查询返回的数据都是按照统计后得到的数字从高到低排序，如果你想要改变这种默认排序方式，比如将每个书籍分类下的 TOP *N* 作者按照该书的销量域（假定我们有个 “sales” 域表示书籍的销量）从高到低进行排序，此时你可以简单地将前面的查询示例进行调整，如下所示：

```
top_authors:{
  type: terms,
  field: author,
  limit: 7,
  sort: "revenue desc",
  facet:{
    revenue: "sum(sales)"
  }
}
```

// 其他部分省略

Subfacet 的类型 type 前面我们已经演示了 terms (即简单地按照域进行 Facet) 和 query (即各种 Query), 此外我们还可以对 range (即区间范围 Query) 创建 Facet 来统计, 示例如下所示:

```
json.facet=sales:{
  type:range,
  field:date_add,
  start:'2016-05-31T22:00:00Z',
  end:'2016-10-31T22:59:59Z',
  gap:'+1DAY',
  facet:{
    ordersSum:'sum(sales_value)'
  }
}
```

此时你可能会有疑问: JSON Facet API 这种方式如何设置请求参数, 其实很简单, json.facet 是固定的参数 key, 后面的整段 JSON 字符串就是参数 value。如果是直接将 URL 输入到浏览器地址里进行测试, 你可以先直接拼接上

```
http://localhost:8080/solr/yourcore/select?q=*&json.facet=.....
```

如果你想要使用 SolrJ 来发送 JSON Facet 这种格式的参数, 你可以类似这样编码, 如下所示:

```
ModifiableSolrParams params = new ModifiableSolrParams();
params.add("json.facet", " 这里是 json 字符串。");
params.set("wt", "wt");
params.set("indent", "true");
QueryRequest query = new QueryRequest(params);
query.setResponseParser(new XXXXXXResponseParser("json"));
NamedList<Object> rsp = solrClient.request(query);
String raw = (String)rsp.get("response");
```

通过上面的示例, 我想你已经感觉到了 Subfacet 的强大, Subfacet 支持对域、任意查询, 任意区间进行统计, 同时还支持 Facet 任意深度的嵌套, 但是, Facet 嵌套是 Pivot Facet 的问题所在。Subfacet 还支持对统计结果进行多种排序, 比如 sum(sales) desc 对指定域求和然后从高到低排序, 然而普通 Facet 仅仅支持 index 和 count 两种排序方式, 而且还不支持反转排序。总而言之, Subfacet 的出现彻底打破了这种僵局。

11.5 Solr Facet Function

传统的 Facet 查询解决了基于某个 Facet 约束条件来统计查询结果集问题, Solr 中扩展传统 Facet 的 Facet Function 支持对 Document 的域进行聚合操作。将 Facet Function 与 Subfacet 功能结合, 两者可以提供强大的实时分析统计能力, 具体请看后续章节的 JSON

Facet API 部分。

11.5.1 聚合函数

Subfacet 将主体信息划分为多个 Facet Bucket，每个 Bucket 又可以嵌套多个 Subfacet，最终会返回每个 Bucket 的详细信息。Solr 提供了 Aggregation Function（聚合函数）用于 Subfacet 的查询统计，如表 11-8 所示。

表 11-8 聚合函数

函 数	使用示例	描 述
sum	sum(sales)	对数值求和
avg	avg(popularity)	对数值求平均值
sumsq	sumsq(rent)	对数值求平方和
min	min(salary)	对数值求最小值
max	max(mult(price,popularity))	对数值求最大值
unique	unique(state)	统计唯一不重复值的总个数（类似 SQL 里的 distinct count(0)）
hll	hll(state)	统计唯一不重复值的总个数，使用 HyperLogLog 算法实现
percentile	percentile(salary,50,75,99,99.9)	用于计算百分比

数值聚合函数，比如 sum、avg，可以作用于数字域或者另外一个函数，也就是说聚合函数也是可以嵌套使用的。不监测计算所有唯一值想要实现 100% 精确度的数据统计几乎是不可能的，但是通常有很多种方法可以用来预算估计。

Facet Function 中的“unique”是 Solr 中用来计算唯一值总数的最快实现。对于单个 Solr 节点，unique 函数总是能够提供确切的数值，对于分布式查询模式下，当每个 Solr 节点上的每个数值不超过 100，那么也可以能够提供确切的数值。当任意给定分片上的唯一值数量超过 100 时，那么统计时会使用下列算法：

- ❑ 每个分片发送 Top 100 结果集以及每个分片的确切的 unique 函数计算的唯一值数量；
- ❑ totalSeen 就是我们能够从所有分片看到的实际结果集数量；
- ❑ uniqueSeen 就是我们能够从所有分片看到的实际唯一值的数量；
- ❑ notSeen 就是所有分片未发送的结果集中的唯一值数量（因为每个分片我们只发送前 100 条结果）；
- ❑ factor=uniqueSeen/totalSeen；
- ❑ 最后估算公式：estimate = uniqueSeen + (notSeen * factor)。

11.5.2 聚合函数与 Subfacet 结合

Facet 查询统计首先会按照 q 参数构建的主查询和 Filter Query 来过滤索引文档，然后我们就可以在剩下的整个索引文档基础之上执行数据统计和分析操作。请看下面这个查询示例：

```
http://localhost:8080/solr/select?q=*&json.facet={x:'avg(price)'}
```

上面的查询示例中我们在 price 域定义了一个名称为 x 的 Facet (维度), 对 price 域求平均值, 这有点类似于 SQL 里的 `select count(0),avg(price) from table_name group by price`。

假如你想要实现每个分类的求和和求平均值等聚合操作, 即类似 SQL 里的 “`select avg(price) as x,sum(price) as y from table_name group by cat`” 功能, 那么你可以像下面这样实现:

```
json.facet={
  categories:{
    type : terms, // 这里的 terms 相当于传统 Facet 里的根据单个域进行 Facet 统计
    field : cat,
    facet:{
      x : "avg(price)",
      y : "sum(price)"}}}
```

11.5.3 Solr 中的 Percentile 函数

Percentile 聚合函数是新 Solr Facet 模块中新增的函数, 它允许每个 Facet Bucket 计算一个百分比 (即 Facet 生成的每个组下的所有索引文档), 甚至可以按照任意指定的 percentile (百分比) 对 Facet Bucket 进行排序。Percentile 聚合函数甚至还支持 Solr 分布式查询, 内部使用的算法是 t-digest, 此算法能够以相对较少的内存消耗提供较高精度的近似值。



注意 Percentile 聚合函数至少需要 Solr 5.1 版本或者更高版本才能使用。

想要使用 Percentile 聚合函数, 首先让我们先创建一个 "percentile" Core 并导入测试数据, 请照例根据随书源码提供的文件创建好 Core 并导入测试数据。

现在让我们使用 Facet 聚合函数来 “切割” 数据! 假设要统计一下我们的工作职位测试数据中, 25%、50%、75% 的工作职位的薪酬大概是多少:

```
http://localhost:8080/solr/percentile/select?q=*&json.facet={
  salary_percentiles : "percentile(salary,25,50,75)"
}
```

返回结果集如下所示:

```
"facets":{
  "count":15,
  "salary_percentiles":[51000.0,74000.0,79500.0]}
```

我们还可以添加其他统计, 比如工作职位的平均薪水、工种总个数、工作地点总个数:

```
http://localhost:8080/solr/percentile/select?q=*&json.facet={
  average_salary : "avg(salary)",
```

```

num_jobs : "unique(job)",
num_states : "unique(loc)",
salary_percentiles : "percentile(salary,25,50,75)"
}

```

返回结果集如下所示:

```

"facets":{
  "count":15,
  "average_salary":63374.6,
  "num_jobs":5,
  "num_states":3,
  "salary_percentiles":[51000.0,74000.0,79500.0]
}

```

现在让我们看看所有工作职位里薪水中间档次对于男女分别是多少:

```

http://localhost:8080/solr/percentile/select?q=*&json.facet={
  by_gender:{terms:{
    field:gender,
    facet:{
      median_salary:"percentile(salary,50)"
    }
  }}
}

```

返回结果如下所示:

```

"facets":{
  "count":15,
  "by_gender":{
    "buckets":[{
      "val":"M",
      "count":8,
      "median_salary":62750.0},
    {
      "val":"F",
      "count":7,
      "median_salary":81000.0}}]}
}

```

我们还可以根据 percentile 聚合函数统计出来的数值进行排序,如果你请求了多个 percentile 值,percentile 聚合函数会返回多个统计数值,此时会按照第一个值进行排序。让我们统计看看 99.9% 的工作职位的薪资对于不同工作地点分别大概是处于什么水平,最后按照 percentile 聚合函数统计出来的每个工作地点的大概薪资进行排序:

```

http://localhost:8080/solr/percentile/select?q=*&json.facet={
  rich_states:{terms:{
    field : loc,
    sort : {sal:desc}, // 你也可以写成 sort:"sal desc"
    facet : {

```

```

        sal : "percentile(salary,99.9)"
    }
  }
}

```

返回的结果如下所示:

```

"facets":{
  "count":15,
  "rich_states":{
    "buckets":[
      {
        "val":"CT",
        "count":5,
        "sal":109909.19600000001},
      {
        "val":"NY",
        "count":5,
        "sal":89438.00000000001},
      {
        "val":"NJ",
        "count":5,
        "sal":80976.0}}]}

```

我们甚至可以执行更有趣的嵌套 Facet 查询, 比如统计一下看看每个工作地点里最挣钱的职业是什么?

```

http://localhost:8080/solr/percentile/select?q=*&json.facet={
  states:{terms:{
    field:loc,
    facet:{
      top_jobs:{terms:{
        field : job,
        sort : "sal desc",
        limit : 1,
        facet:{
          sal : "percentile(salary,99.9,50,10)"}}}} }}}

```

返回的结果集太长了, 这里就省略不贴了。我们甚至还可以统计每个工作点的工作职位的中间档薪酬随着时间推移的变化趋势:

```

http://localhost:8080/solr/percentile/select?q=*&json.facet={
  states:{terms:{
    field:loc,
    facet:{
      over_time:{range:{
        field : year,
        start : 2011,
        end : 2015,
        gap : 1,
        facet:{

```



```

        median_salary : "percentile(salary,50)"
    }
  }
}
}
}
}

```

到此我想你已经感受到了 Percentile 聚合函数的强大了，请尽情发挥你自己的想象力，并结合实际项目需求构造出更复杂的统计查询。

11.6 JSON Facet API

上一小节中我们已经学习了 Subfacet，你会发现 Subfacet 的使用语法完全依赖于 JSON Facet API，正由于 JSON Facet API 相比于传统的 key-value 键值对这种指定参数形式更灵活且更有层级感，从而使得你能够借助 JSON Facet API 构建出更复杂更强大的 Subfacet 查询统计。前面的 Subfacet 小节中我们多多少少演示了 JSON Facet API 的使用语法，大家应该对 JSON Facet API 有了初步的了解，接下来再系统性学习 Solr 中强大的 JSON Facet API。


11.6.1 JSON Facet API 简介

Solr 5 采用 JSON API 这种结构完全重写了 Facet 重写功能以及数据分析模块，从而更加灵活地控制 Facet 和 Analytic 命令。

采用 JSON 这种结构特性来实现 Subfacet 的嵌套比传统查询请求以扁平式结构指定请求参数显得更加自然且更加灵活。采用 JSON Facet API 新设计的 Facet 模块有以下几点目标：

- 完美的 JSON 支持；
- 对于复杂的嵌套 Facet 命令采用 JSON 格式表示使得程序结构更简单；
- 支持更标准化的响应数据输出格式，简化客户端数据解析；
- 完美的数据分析功能支持；
- 支持根据任意的计算值对 Facet Bucket 进行排序；
- 支持以更轻便的方式来执行分布式 Facet；
- 支持与 Solr 其他查询功能更好的集成度。

当然，如果你更喜欢使用之前的 Facet 功能，也是可以的，只要你乐意，甚至可以同时使用两者。

 **注意** JSON Facet API 现在是 JSON Request API 的一部分，因此一个完整的 Solr 请求都可以采用 JSON 格式来表示。

11.6.2 JSON Facet 简单使用

新设计的 JSON Facet 与生俱来就拥有着传统 Solr Facet 所没有的简单易用的增强型嵌套式 JSON 结构语法。以下 Solr 中的 Range Facet 命令分别采用传统 Facet 和增强型 JSON Facet 两种写法：

```
// 传统 Facet 写法
&facet=true
&facet.range={!key=age_ranges}age
&f.age.facet.range.start=0
&f.age.facet.range.end=100
&f.age.facet.range.gap=10
&facet.range={!key=price_ranges}price
&f.price.facet.range.start=0
&f.price.facet.range.end=1000
&f.price.facet.range.gap=50
```

以下是采用 JSON Facet API 重写的 Range Facet 命令的等价写法：

```
{
  age_ranges: {
    type : range
    field : age,
    start : 0,
    end : 100,
    gap : 10
  },
  price_ranges: {
    type : range
    field : price,
    start : 0,
    end : 1000,
    gap : 50
  }
}
```

传统 Facet 并不能很好地支持嵌套 Facet，然而，你也已经看到了，JSON Facet API 是如何完美地支持嵌套 Facet（即 Subfacet）的，随着不断深入学习使用 Subfacet，你会越发清晰地感觉到 JSON Facet API 太好用了。

为了使采用 JSON Facet API 构造的 Facet 命令更清晰、更直观、阅读性更强，Solr 对 JSON Facet API（准确来说，是对 JSON Request API）扩展了 JSON 注释，具体如下所示：

```
{ // 这是 JSON 中的单行注释
  /* 同样支持传统的 C 语言风格注释 */
  x : "avg(price)", // 简单的字符串其实可以不用在两头添加引号
  y : 'unique(manu)' // 字符串也可以使用单引号
}
```

缩进良好的 JSON 会更容易阅读, 如果你有一个很大段的 JSON 字符串没有缩进, 又试图要理解它, 这时你可以将其粘贴到在线的 JSON 格式化网站中进行美化, 比如如下这个网站链接: <http://jsonformatter.curiousconcept.com>。

11.6.3 Facet 类型

JSONFacet 大致有两种类型 Facet: 一种是将 Facet domain 划分成多个 Facet Bucket; 另一种是 Facet 聚合函数, 它能够统计每个 Facet Bucket 下的所有索引文档的信息 (比如计算总和或者求平均值)。

Facet domain 是一系列值的集合 (通常是通过索引文档集合来定义), 统计计算是针对 Facet domain 进行的。Root domain (根 domain) 必须是 q 参数的主查询或者任意的 Filter Query 匹配的所有索引文档集合。

对于任意的 Facet 命令, 你可以在 Facet 统计计算之前, 使用 domain 关键字去修改 Facet 的 domain:

- ❑ blockParent 和 blockChildren 一般用于嵌套 Document 中在 parent 和 children 之间去切换 domain;
- ❑ excludeTags 用于新版 JSON Facet 的 Multi Select Facet 中排除部分被标记的 Filter Query。

Solr 中的 Statistics (数据分析) 也完全集成到 Facet 中, 首先根据主查询和 Filter Query 定义了一个 Facet domain, 然后在 Facet domain 下创建了一个 Facet Bucket, 然后可以继续再 Facet Bucket 下定义新的 Facet domain, 在新的 Facet domain 下创建子 Facet Bucket, 如此嵌套循环。我们甚至能够在 JSON Facet 将其划分为更多 Facet Bucket 之前, 在顶级 Facet Bucket 中进行 statistics (数据分析), 如下所示:

```
json.facet={
  x : "avg(price)",
  y : "unique(manufacturer)"
}
```

Solr 中的 Statistics (数据分析) 需要使用大量的聚合函数, 有关 Facet 中的聚合函数请翻到前面的章节进行阅读回顾。

11.6.4 JSON Facet 语法

JSON Facet 语法的通用格式如下:

```
<facet_name> : { <facet_type> : <facet_parameter> }
```

<facet_name> 即你定义的 Facet (即维度) 名称, 名称可自定义 <facet_type> 表示 Facet 查询的类型, 可选值有 terms、query、range。<facet_parameter> 表示一些 Facet 参数, 可以

有多个，示例如下所示：

```
top_authors : { terms : { field : authors, limit : 5 } }
```

自 Solr5.2 版本开始，你可以使用 type 属性来定义 Facet 查询的类型，使得整个 JSON Facet 表示结构显得更扁平化，语法格式如下所示：

```
<facet_name> : { "type" : <facet_type> , <other_facet_parameter(s)> }
```

使用示例如下所示：

```
top_authors : { type : terms, field : authors, limit : 5 }
```

11.6.5 Term Facet

Term Facet 或者 Field Facet 会根据域的唯一值中生成一个 Facet Bucket，且该域必须是 indexed=true 且 docValues=true 的域。注意，这里说的 Term Facet 都指的是新版 JSON Facet 下的 Term Facet。以下是 Term Facet 的简单示例：

```
{
  top_genres : {
    type : terms,
    field : genre_field,
    limit : 3,
    mincount : 2
  }
}
```

表 11-9 列此举了 Term Facet 支持的所有参数。

表 11-9 Term Facet 参数表

参 数	描 述
field	指定你需要根据哪个域进行 Facet
offset	用于分页中，表示跳过前 N 个 Facet Bucket，默认值为零
limit	限制返回的 Facet Bucket 的数量，默认值为 10
mincount	表示只返回匹配的索引文档总数 count 值不小于 minCount 参数指定数量的 Facet Bucket。默认值为 1
sort	用于指定如何对 Facet Bucket 进行排序。count 表示按照匹配的索引文档数量排序，index 表示按照 Facet Bucket 值的自然顺序（即比较 ASCII 码值）排序，你还可以按照当前 Facet Bucket 中应用的任意 Facet Function 的计算值进行排序，默认值是 "count desc"。sort 参数同时还支持以 JSON 格式进行指定，比如 sort:{count:desc}。排序顺序可以是 asc（升序）或者 desc（降序）
missing	一个 boolean 参数，用于指定对应域值为空值的 Document 定义的 Facet Bucket 是否应该被返回，默认值为 false 即不返回
numBuckets	一个 boolean 参数，表示是否将 numBuckets 属性添加到响应结果中，返回的 numBuckets 属性值表示 Facet 统计得到的 Bucket 总个数（注意：不是实际返回的 Bucket 个数）。默认值为 false 即不返回 numBuckets 属性

(续)

参 数	描 述
allBuckets	一个 boolean 参数, 如果 allBuckets=true, 那么会将 allBuckets 作为属性添加到响应结果中。allBuckets 属性值表示所有的 Bucket 的并集。对于多值域 (即 multiValued=true 的域), 返回的 allBuckets 并不是表示当前 Facet domain 下的所有匹配的索引文档数量, 因为单个索引文档可能会属于多个 Facet Bucket。默认值为 false
prefix	只有以 prefix 参数指定的字符前缀开头的 Facet Bucket 才会被返回
method	<p>用于指定如何对域执行 Facet</p> <p>method:uif, 表示 UninvertedField, 一种索引方式, 对于多值域会使用顶级数据结构为 NRT 近实时搜索性能提供优化。</p> <p>method:dv, 表示 DocValues, 一种索引方式, 多值域会使用每个段文件的数据结构, 这个 method 反映着 Facet 作用于真实的 DocValues 域上, 但是实际却作用于 Java 堆内存中构建的 DocValue 上, 如果内存中 docValues 不存在, 那么会从硬盘索引中即时加载并写入到堆内存中的 DocValues。这个 method 比较适合适用于快速更新索引的场景下。</p> <p>method:stream, 当以流式返回响应给请求端时, 这个 method 会临时创建每个 Facet Bucket (包括任意的子 Facet)。当前仅仅只支持 sort=index</p>

11.6.6 Query Facet

Query Facet 会根据指定的 Query 生成单个匹配该 Query 的 Facet Bucket。以下是一个 Query Facet 的简单使用示例:

```
{
  high_popularity : {
    type : query,
    q : "popularity:[8 TO 10]",
    facet : { average_price : "avg(price)" }
  }
}
```

11.6.7 Range Facet

Range Facet 会针对数字域或者 date 域生成多个 Range Bucket。以下是一个 Range Facet 的简单使用示例:

```
{
  prices : {
    type : range,
    field : price,
    start : 0,
    end : 100,
    gap : 20
  }
}
```


为了迁移更方便, JSON Facet 中的参数名称、参数值、参数语义都是直接从 Solr 传统

的 Range Facet（即非新版的 JSON Facet）中照搬过来的，如表 11-10 所示。

表 11-10 JSoN Facet 参数表

参 数	描 述
field	根据指定的数字域或者 date 域生成 Range Bucket，这里的 field 表示该域的域名称
mincount	表示 Bucket 匹配的索引文档总个数最小值必须是 mincount，否则该 Bucket 不会被返回，默认值为零
start	表示 Range 区间范围的上边界值
end	表示 Range 区间范围的下边界值
gap	自动划分 Facet Bucket 时取的区间间隔值
hardend	这是一个 boolean 参数，如何 hardend=true 则意味着最后一个 Facet Bucket 的区间下边界值就等于 end 参数设置的值，即便它小于 gap 参数设置的间隔值。如果 hardend=false，则意味着最后一个 Facet Bucket 的实际区间下边界值等于 gap 的宽度，实际可能会超过 end 参数的下边界值
other	other 参数表示除了统计 [start,end] 区间内的数量之外，还需要统计其他区间的值 before：表示需要对 start 之前的日期做个统计 after：表示需要对 end 之后的日期做个统计 between：表示需要对 start 与 end 之间的日期做个统计 none：表示不做任何汇总统计 all：表示 before,after,between 都需要做统计
include	默认 start 和 end 参数表示的区间是包含 start 上边界值但是不包含 end 这个下边界值。include 参数用于指定什么情况下是否应该包含上边界值和下边界值，可选参数值有：lower、upper、edge、outer、all。具体各个参数的含义与传统的 Range Facet 中的 facet.range.include 参数表达的含义一致，请翻到第 6 章第 4 节查看该参数详细说明，这里就不再重复了

所有的 JSON Facet 都有一个通用参数——domain，用于在 Facet 执行之前改变 Facet 命令的 domain。对于 Multi Select Facet 和 Nested Document(嵌套索引文档)Facet 使用场景下，切换 Facet domain 会显得非常有用。

 **注意** 如果你使用的是 Solr5.2 或 Solr5.3，那么你需要使用 `excludeTags:mytag, domain:{excludeTags:mytag}` 方式只适用于 Solr5.4 及更高的版本。

11.6.8 Multi-Select Facet

Multi-Select Facet 是一种强大的 Facet 类型，它允许用户在某个 Facet（维度）下查看和选择多个 Facet 值。我们在之前的第 6 章已经学习过了 Solr 中的 Multi-Select Facet，这次我们采用新版的 JSON Facet API 来重新实现 Multi-Select Facet。你需要在我们开始讲解之前，提前创建好 "multiselect" Core 并执行测试类 IndexMultiSelect 导入测试数据。

假设有 Size、Color、Brand 这 3 个 Brand，之所以称其为 multi-select（多选）Facet，

是因为对于 Color 和 Brand 这两个 Facet (维度), 我们希望用户能够多选, 而对于 Size 这些 Facet, 我们假设用户每次只对一种尺码感兴趣。假如以下就是我们想象中的用户搜索 “running shorts” 之后的用户 UI 界面:

```

=== Size ===    === Color ===    === Brand ===
[Small] (7)     [ ] Red (2)       [ ] Nike (7)
[Medium] (5)    [ ] Blue (8)      [ ] Adidas (5)
[Large] (6)     [ ] Green (3)     [ ] Reebok (4)
[ ] Black (5)   [ ] Under Armour (2)
(返回的结果就在下方按照人气值从高到低显示, 请充分发挥你的想象力 ...)
```

请注意 Color 和 Brand 这两个 Facet 前面有 Checkbox, 用来表示哪个条件被选中了, 刚开始用户还没有勾选, 所以都没有被选中。

假设用户此时想要选择 “Blue”, 错误的做法是 (返回界面如图 11-4 所示):

```

http://localhost:8080/solr/multiselect/select?q=*:*&fq=color:Blue
&json.facet={
  sizes:{type:terms, field:size},
  colors:{type:terms, field:color},
  brands:{type:terms, field:brand}}
```

```

=== Size ===    === Color ===    === Brand ===
[Small] (817)    [ ] Red (0)       [ ] Nike (609)
[Medium] (803)  [X] Blue (2411)   [ ] Adidas (629)
[Large] (791)   [ ] Green (0)     [ ] Reebok (610)
[ ] Black (0)   [ ] Under Armour (563)
(Top N条color=blue的索引文档会显示在这下方)
```

图 11-4 选择 “Blue” 之后的 UI 界面

仿佛哪里不对劲? 现在 Size 和 Brand 这两个 Facet 显示的都是 Color=Blue 条件下的统计数字, 这些数字是正确的并且也是我们所想要的, 而且在下方也显示了 Color=Blue 的 TOP N 索引文档。因为我们过滤掉所有 Color 不等于 Blue 的索引文档, 所以我们的 Color 中其他颜色的 Facet 统计数字都是零, 只有 Blue 这一个颜色有统计数字。但是我们仍然希望显示其他颜色的统计信息, 这样用户在选择了 “Blue” 颜色之后, 还能够继续挑选其他颜色, 然而目前的情形是, 其他颜色下的统计数字显示都是零, 用户无法继续按照其他颜色进行过滤了。

此时, 正确的做法如下:

```

http://localhost:8080/solr/multiselect/select?q=*:*&fq={!tag:COLOR}color:Blue&
json.facet={
  sizes:{type:terms, field:size},
  // Solr5.4 之前的版本, 请使用 excludeTags:COLOR 替代 domain:{excludeTags:COLOR}
```

```
// 因为 domain 属性是 Solr5.4 版本之后才开始支持的
colors:{type:terms, field:color, domain:{excludeTags:COLOR} },
brands:{type:terms, field:brand}
}
```

然后返回的结果界面如 11-5 所示。

```
=== Size ===      === Color ===      === Brand ===
[Small] (817)    [ ] Red (2496)    [ ] Nike (609)
[Medium] (803)  [x] Blue (2411)  [ ] Adidas (629)
[Large] (791)   [ ] Green (2492) [ ] Reebok (610)
                [ ] Black (2601) [ ] Under Armour (563)
(Top N条color=blue的索引文档会显示在这下方)
```

图 11-5 使用 JSON Facet API 实现 Multi-select Facet 后的用户界面

在 JSON Facet API 中, domain 关键字通常用于在 Facet 统计计算之前改变 Facet 的 domain (说白了就是改变 Facet 的作用对象)。通过 excludeTags 属性可以用来排除作用于当前 Facet 的 Filter Query。在上面的示例中,我们为 Color 这个 Facet 指定了 domain:{excludeTags:COLOR}, 表示被标记为 "COLOR" 的 Filter Query 将不再作用于当前 Facet。

我们在第 6 章已经学习了 Filter Query (即 fq 参数) 可以使用 {!tag=mytag} 语法进行标记, 比如 fq={!tag=COLOR}color:Blue, 而且多个 Filter Query 可以被标记为同一个 tag 名称, 比如:

```
fq={!tag=foo}one_filter&fq={!tag=foo}another_filter
```

当然单个 Filter Query 也可以打多个 tag (标记), 比如:

```
fq={!tag=tag1,tag2,tag3}my_field:my_filter
```

在 JSON Facet API 中, 也可以实现类似的功能, 不过此时使用的 domain 和 excludeTags 属性的组合, 比如:

```
colors:{type:terms, field:color, domain:{excludeTags:COLOR}}
```

与传统的 Facet 类似, excludeTags 也可以指定多个 tag 名称, 比如:

```
colors:{type:terms, field:color, domain:{excludeTags:COLOR1,COLOR2,COLOR3}}
```

为 excludeTags 指定多个 tag 名称除了可以使用逗号分隔多个 tag 之外, 还可以以 JSON 数组的形式来表示, 比如:

```
colors:{type:terms, field:color, domain:{excludeTags:["COLOR1","COLOR2","COLOR3"]}}
```

对于 Subfacet 而言, excludeTags 可以应用于任意层级的 Facet Bucket 中。如果一个父

Facet 拥有一个 `excludeTags:tag1`，并且子 Facet 想要额外排除一个被标记为 "tag2" 的 Filter Query，那么子 Facet 需要显式指定 `excludeTags:"tag1,tag2"`。

11.7 Interval Facet

Solr 中支持另外一种 Facet 形式：Interval Facet。它跟 Range Facet 很相似，但更像是 Facet Query，它允许设置变量间隔并统计在指定域的值间隔内的索引文档总数量。尽管这样的功能我们可以通过使用 Facet Query 加 Range Query 实现，但是 Interval Facet 更适合于为同一个域设置多个值间隔，而 Facet Query 更适合于充分利用 Filter 缓存使得查询更高效，Interval Facet 默认会使用 `docValues` 来保证性能，当然前提是该域的 `docValues=true`，如果该域的 `docValues=false`，那么就会使用 `fieldCache`（域缓存）。

Interval Facet 拥有以下 2 个配置参数（如表 11-11 所示）。

表 11-11 Interval Facet 配置参数表

参 数	描 述
<code>facet.interval</code>	指定 Interval Facet 作用于哪个域上，此参数可以被指定多次，比如 <code>facet.interval=price&facet.interval=size</code>
<code>facet.interval.set</code>	设置值间隔，此参数可以设置多次。注意此参数设置是全局生效的，即作用于所有 <code>facet.interval</code> 设置的域上，如果你想要单独为某个域设置，那么你需要使用这种语法进行设置： <code>f.<fieldname>.facet.interval.set</code> 示例： <code>f.price.facet.interval.set=[0,10]&f.price.facet.interval.set=(10,100]</code>

设置 `facet.interval.set` 参数是需要遵循一定语法的，指定的间隔值需要以（或 [字符开头然后是 start 起始值，紧接着是一个逗号，逗号后面是 end，最后以）或] 字符结尾。这就跟数学里的区间表示法类似。比如：

`(1,10)` -> 表示 1 到 10 之间的数字，但是不包含两个边界值。

`[1,10)` -> 表示 1 到 10 之间的数字，包含 1 但是不包含 10。

`[1,10]` -> 表示 1 到 10 之间的数字，包含 1 和 10。

`[1, *]` -> 表示 ≥ 1 的数字。

`[*, *]` -> 表示所有数字。

`facet.interval.set` 参数还适用于 String，但是字符串两头可以不用加引号。比如：

```
[Buenos Aires,New York]
```

间隔值如果是 String，在比较时是区分大小写的，且间隔值中间如果有 “([)]” 等特殊字符时，你可以使用 “\” 斜杠字符进行转义。注意：间隔值两头的空格字符会被自动剔除，间隔值 start 不能大于 end，但是 start 可以等于 end，start=end 等于按照指定值进行 Facet，比如 `[A,A]`，`[B,B]`，`[C,Z]`。

Interval Facet 也支持对输出 key 进行重命名, 示例如下所示:

```
&facet.interval={!key=popularity}some_field
&facet.interval.set={!key=bad}[0,5]
&facet.interval.set={!key=good}[5,*]
&facet=true
```

11.8 Hierarchical Facet

对于简单的分类, 我们使用 Field Facet 就可以搞定, 但是对于分层次结构化的路径表达式数据, 如何有效地进行 Facet 是一个值得探讨的问题。我们知道, 文件系统的文件路径通常是分多个层次的, 比如:

```
/opt/git-space/solr-book/example-docs/ch11/cores/multiselect
```

如上的示例所示, 一个文件路径会使用反斜杠 (这里以 Linux 系统环境为例) 划分 N 个层级, 每个层级应该都有个层级深度。假设我们需要对这些文件路径进行索引, 你可能会这样索引:

```
Doc#1: NonFic > Law
Doc#2: NonFic > Sci
Doc#3: NonFic > Hist, NonFic > Sci > Phys
```

上面的测试数据表示 NonFic 目录下有 Law、Sci、Hist 等 3 个目录, 而 Sci 下有一个 Phys 目录。为了表示每个目录所处深度, 我们可以将数据转变为下面这种格式进行索引:

```
Doc#1: 0/NonFic, 1/NonFic/Law
Doc#2: 0/NonFic, 1/NonFic/Sci
Doc#3: 0/NonFic, 1/NonFic/Hist,
      0/NonFic, 1/NonFic/Sci, 2/NonFic/Sci/Phys
```

这里我们将每个路径表达式根据目录深度拆分为多个子路径表达式, 同时每个表达式前面添加一个表示目录深度的数字。为了便于后面的查询演示, 你创建 "hierarchical-facet" Core 以及导入测试数据 (请在随书源码中查找相关配置文件、测试数据以及测试类)。

假设我们想要对 category 域进行 facet 同时限制 facet.prefix = 1/NonFic, 即统计 NonFic 目录下有几个子目录, 查询示例如下所示:

```
http://localhost:8080/solr/hierarchical-facet/select?q=*:*&
facet.field=category&facet=true&
facet.prefix=1/NonFic&
facet.mincount=1
```

返回的结果如下所示:

```
"facet_fields":{
  "category":[
```

```
"1/NonFic/Sci",2,
"1/NonFic/Hist",1,
"1/NonFic/Law",1]],
```

但是此时如果我只想统计 1/NonFic/Sci 目录下的子目录个数，我们可以简单的添加一个 fq 来过滤即可，查询示例如下所示：

```
http://localhost:8080/solr/hierarchical-facet/select?q=*&
facet.field=category&facet=true&
fq={!raw f=category}1/NonFic/Sci&
facet.prefix=2/NonFic/Sci&
facet.mincount=1
```

返回结果如下所示：

```
"facet_fields":{
  "category":[
    "2/NonFic/Sci/Phys",1]],
```

没错，我们确实可以使用已经学习到的 Facet 知识解决关于多层级路径的 Facet 统计查询问题，但是，我们可以选择另外一种实现方式，那就是：Hierarchical Facet（分层 Facet）。其实主要还是使用 PathHierarchyTokenizerFactory 这个 Tokenizer 工厂创建的 PathHierarchyTokenizer 帮我们自动将类似：

```
Doc #1: /usr/local/apache
Doc #2: /etc/apache2
Doc #3: /etc/apache2/conf.d
```

这样的数据自动转化成如下格式进行索引：

```
Doc #1: /usr, /usr/local, /usr/local/apache
Doc #2: /etc, /etc/apache2
Doc #3: /etc, /etc/apache2, /etc/apache2/conf.d
```

即将单个路径划分为多个层次的路径，跟我们前面自己实现的方式不同的是，PathHierarchyTokenizer 是将划分出来的子路径表达式作为顶级路径表达式的同义词来处理，比如上面的 Doc1 索引文档中的 /usr/local,usr/local/apache 其实都是 /usr 的同义词（即分出来的 Token 的 position 属性值全都相同），同时，每个被索引的子路径表达式前面还没有表示深度的数字，这跟我们前面自己的实现方式略微有所不同，实际 Facet 查询还是使用传统的 Facet 查询方式。若改成 Hierarchical Facet 这种方式实现，我们的 schema.xml 中的 category 域的域类型需要进行修改，如下所示：

```
<fieldType name="text_path" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.PathHierarchyTokenizerFactory" delimiter="/" />
  </analyzer>
</fieldType>
```

```
<!-- 对 category 域进行重新命名 -->
<field name="category" type="string" indexed="true" stored="true" multiValued="true"/>
-->
<field name="category" type="text_path" indexed="true" stored="true"
multiValued="true"/>
```



注意 上面的 `text_path` 域类型在配置 `TokenizerFactory` 时, `analyzer` 元素的 `type` 属性值请设置为 `index`, 即只在索引时使用 `PathHierarchyTokenizer` 进行分词, `query` 查询时不需要。

11.9 Solr Stats 组件

Solr 中的 Stats 组件可以对 `numeric`、`string`、`date` 域进行简单的数据统计, 类似关系型数据库中的 `min`、`max`、`avg`、`count`、`sum` 等功能。

本章中涉及的示例查询需要你提前创建好 "techproducts" Core, 请从随书源码中获取 Core 的配置文件并运行导入数据测试类 `IndexStats` 进行数据索引。

以下这个查询示例我们根据 `price` 和 `popularity` 这两个域以及 "memory" 这个 Term 在 `text` 域上出现的频率即 `termfreq` 函数计算得到的值进行分别统计:

```
http://localhost:8080/solr/techproducts/select?q=*&wt=json&stats=true&stats.
field={!func}termfreq('text','memory')&stats.field=price&stats.field=popularity&ro
ws=0&indent=true
```

返回的结果集部分如下所示:

```
"stats_fields":{
  "termfreq('text','memory')":{
    "min":2.0,
    "max":2.0,
    "count":3,
    "missing":0,
    "sum":6.0,
    "sumOfSquares":12.0,
    "mean":2.0,
    "stddev":0.0},
```

其中 `min` 表示最小值, `max` 表示最大值, `count` 表示匹配的索引文档总个数, `missing` 表示域值为空值的索引文档总个数, `sum` 表示和, `sumOfSquares` 表示平方和, `mean` 表示平均值, `stddev` 表示标准差。

并不是所有的统计项都支持对所有类型的域进行统计, 也并不是所有统计项默认都是开启。下面表格详细解释了 Solr Stats 组件支持的统计项, 如表 11-12 所示。

表 11-12 Solr Stats 组件支持表

参 数	描 述	支持的域类型	默认开启
min	表示指定域或函数的最小值	所有域类型	YES
max	表示指定域或函数的最大值	所有域类型	YES
sum	表示每个索引文档在指定域或函数上的和值	数字域和 date 域	YES
count	表示指定域或函数上的索引文档总数	所有域类型	YES
missing	表示指定域的域值或函数计算值为空值的索引文档总数	所有域类型	YES
sumOfSquares	表示指定域或函数的平方和	数字域和 date 域	YES
mean	表示指定域或函数的平均值	数字域和 date 域	YES
stddev	表示指定域或函数的标准差	数字域和 date 域	YES
percentiles	通过指定的界点统计每个区间内的近似值，近似值采用的是 t-digest 算法计算	数字域	NO
distinctValues	表示指定域的域值或函数计算值上唯一不重复值的总数，此统计项计算执行开销很大。	所有域类型	NO
countDistinct	表示指定域的域值或函数计算值上唯一不重复值的精确总个数，此统计项执行计算开销很大	所有域类型	NO
cardinality	用于对指定域或者函数计算距离近似值，采用 HyperLogLog 算法实现。它比 countDistinct 性能要高效，但是并不是 100% 精确值，开启此统计项可以输入 true 或者是一个 [0, 1] 之间的 float 数值，0.0 表示尽量少占用内存，1.0 表示尽量使用更多的内存来保证计算精确度。如果设置为 true，那么默认精度为 0.3	所有域类型	NO

Solr Stats 组件可以接收以下参数（如表 11-13 所示）。

表 11-13 Solr Stats 组件接收参数表

参 数	描 述
stats	表示是否开启 stats 组件，默认值 false
stats.field	表示在哪个域上执行统计，此参数可以指定多次
stats.facet	在指定的 Facet 内执行 stats 统计，这是一个高级配置参数，你可以理解为 stats.field + facet.pivot
stats.calcdistinct	如果此参数设置为 true，那么 countDistinct 和 distinctValues 两个统计项会被返回，此统计计算开销很大，所以默认它是被禁用的。此参数可以为每个域单独设置，比如 f.<field>.stats.calcdistinct=true

与 Facet 查询组件类似，Stats 组件的 stats.field 参数也支持部分 Local params，比如：

```
// 排除某个 Filter Query 对 Stats 统计的影响
stats.field={!ex=filterA}price
// 表示重命名 stats 统计返回的 key
stats.field={!key=my_price_stats}price
// 你也可以单独设置某个域返回的统计项
stats.field={!min=true max=true percentiles='99,99.9,99.99'}price
```

尽管 stats.facet 参数已经不建议使用，但是你可以通过在 stats.field 参数上使用 !tag 语

法来标记 stats 统计, 然后可以在 facet.pivot 中通过 !stats 语法引用 stats.field 参数中定义的 tag, 从而实现将一个 stats 统计引入到一个 Facet (维度) 中。具体请执行以下查询示例:

```
http://localhost:8080/solr/techproducts/select?q=*:*&
wt=json&indent=true&stats=true&
stats.field={!tag=piv1 min=true max=true}price&
stats.field={!tag=piv2 mean=true}popularity&
facet=true&
facet.pivot={!stats=piv1}cat,inStock&
facet.pivot={!stats=piv2}manu,inStock
```

上面的查询示例表示在 "cat,inStock" 这个 pivot 中嵌入 tag=piv1 的 stats 统计, 在 "manu,inStock" 这个 pivot 中嵌入 tag=piv2 的 stats 统计。

11.10 Solr Terms 组件

Solr 中的 Terms 组件允许访问指定域中被索引的 Term 以及每个 Term 匹配的索引文档总个数。当想要实现 Autosuggest (自动拼写建议) 功能时可能会有用。按照索引顺序来获取 Term 信息的执行过程是很快的, 因为它内部事先直接使用的是 Lucene 中的 TermEnum 来迭代整个 Term 字典。

从某种意义上来讲, 这个查询组件提供了一种快速的对索引文档域级别的查询访问, 而不是通过 Query 或者 Filter Query。返回的 Term 在文档中出现的频率值其实是包含已经标记为删除但是尚未实际从硬盘索引文件中删除的索引文档的。

如果你想要启用 Solr 中的 Terms 组件, 首先需要在 solrconfig.xml 中进行如下配置:

```
<searchComponent name="terms" class="solr.TermsComponent"/>
<requestHandler name="/terms" class="solr.SearchHandler" startup="lazy">
  <lst name="defaults">
    <bool name="terms">true</bool>
    <bool name="distrib">false</bool>
  </lst>
  <arr name="components">
    <str>terms</str>
  </arr>
</requestHandler>
```

上面的 request handler 的 defaults 参数配置中, terms 参数表示是否开启 Terms 组件, distrib 参数是否开启分布式环境下的 Terms 查询支持, 默认只支持单个 Core 下的 Terms 组件查询。

表 11-14 列举了 Terms 组件支持的所有参数, 你可以通过指定这些参数来控制 Terms 组件返回哪些 Terms。也可以将它们直接配置在 request handler 的 defaults 参数部分, 即将它们当作默认值不需要每次都手动设置。

表 11-14 Terms 组件支持参数表

参 数	是否必需	默认值	描 述
terms	否	false	表示是否启用 Terms 组件，默认值为 false 即不开启
terms.fl	是	null	用于指定从哪个域上获取 Term 信息，示例：terms.fl=title
terms.list	否	null	返回指定的多个 Term 在索引文档中的出现频率，Term 始终以在索引中的顺序返回
terms.limit	否	10	指定最多返回多少个 Term，默认值为 10，如果此参数设置为小于 0 的数字，那么即表示对返回的 Term 数量不做限制。尽管此参数不是必需的，但是此参数或者 terms.upper 参数至少应该任意定义一个
terms.lower	否	空字符串	表示从 terms.lower 参数指定的 Term 开始返回，如果索引中的 Term 小于 terms.lower 参数指定的 Term，那么该 Term 将不返回。示例：terms.lower=orange
terms.lower.incl	否	true	表示是否包含 terms.lower 参数指定的 Term
terms.mincount	否	null	指定 Term 在索引文档中出现的最小次数 (\geq mincount)，小于此参数设置的次数，那么该 Term 将不会返回
terms.maxcount	否	null	指定 Term 在索引文档中出现的最大次数 (\leq maxcount)，大于此参数设置的次数，那么该 Term 将不会返回。默认值为负 1
terms.prefix	否	null	表示以指定前缀开头的 Term 才返回，示例：terms.prefix=ang
terms.raw	否	false	如果此参数设置为 true，那么会返回索引中该 Term 实际索引存储值，而不关心人类是否可以阅读。比如，索引中的数字实际索引方式人类是无法阅读的
terms.regex	否	null	表示只有符合指定正则表达式的 Term 才会被返回，示例：terms.regex=*pedist
terms.regex.flag	否	null	<p>当使用 terms.regex 参数匹配 Term 时，可以通过 terms.regex.flag 参数来设置 Java 中正则表达式的标记，比如是否忽略大小写。是否支持匹配多行等。此参数可选值有：</p> <ul style="list-style-type: none"> case_insensitive comments multiline literal dotall unicode_case canon_eq unix_lines <p>示例：terms.regex.flag=case_insensitive</p>
terms.stats	否	null	表示是否返回结果集的统计信息，当前只返回 numDocs 属性。如果 terms.list 参数同时也配置了，此时还会提供 Term 的 idf 信息
terms.sort	否	count	<p>定义如何对返回的 Term 进行排序，可选值有：count、index</p> <p>count：表示 term 在索引文档中的出现频率从高到低排序</p> <p>index：表示按照 Term 实际在索引文档中的索引顺序排序</p>
terms.upper	否	null	表示大于 terms.upper 参数指定的 Term 的 Term 不会被返回，示例：terms.upper=plum
terms.upper.incl	否	false	表示是否包含 terms.upper 参数指定的 Term，默认值为 false

下面是一个 Term 组件查询的简单示例：

```
http://localhost:8080/solr/techproducts/terms?wt=json&indent=true&terms.fl=name
```

返回的结果集如下所示：

```
"terms":{
  "name":[
    "184",3,
    "1gb",3,
    ...// 省略
    "system",3]]}
```

如果 Solr 中的 Suggester 不能满足你的需求，你可以使用 Terms 组件来实现相似的功能。比如用户输入“at”，此时使用 Terms 组件我们自己查询出索引中以 at 开头的所有 Term，按照出现频率从高到低排序，并限制返回前 10 个，查询示例如下所示：

```
http://localhost:8080/solr/techproducts/terms?wt=json&indent=true&
terms.fl=name&terms.prefix=at&terms.sort=count&terms.limit=10
```

Terms 查询组件还支持分布式查询，此时你需要指定 shards 和 shards.qt 参数，shards 参数表示你需要在哪些 shards 上的索引文档执行 Terms 组件查询，shards.qt 参数用于指定 Solr 请求某个 Shard 需要用到的 Request Handler。

11.11 SolrTerm Vector 组件

Solr 中的 Term Vector (Term 向量) 组件首先它是一个查询组件，是用来返回查询匹配的结果集的一些额外信息，比如：Term 向量信息、Term 在文档中的出现频率，文档中包含 Term 的频率、Term 的 position 位置信息、Term 的偏移量信息。当你在结果集中返回这些信息，你可能会需要使用 Term Vector 组件。

Term Vector 组件在 Solr 中默认是不开启的，你需要显式的在 solrconfig.xml 中配置 Term Vector 组件并开启它，配置示例如下所示：

```
<searchComponent name="tvComponent"
  class="org.apache.solr.handler.component.TermVectorComponent"/>
<requestHandler name="/tv" class="org.apache.solr.handler.component.
SearchHandler">
  <lst name="defaults">
    <bool name="tv">true</bool>
    <str name="wt">json</str>
    <bool name="indent">true</bool>
  </lst>
  <arr name="last-components">
    <str>tvComponent</str>
  </arr>
</requestHandler>
```

然后你需要为你的某个域开启 `termVectors` 属性, 如果还需要返回该域的 Term 的位置信息或者偏移量信息, 那么你需要添加 `termPositions` 和 `termOffsets` 属性, 配置示例如下所示:

```
<field name="includes"
  type="text_general"
  indexed="true"
  stored="true"
  multiValued="true"
  termVectors="true"
  termPositions="true"
  termOffsets="true" />
```

然后请重新加载 Core 使其立即生效, 同时从随书源码中找到 `IndexTermVector` 测试类, 导入测试数据, 为接下来的 Term Vector 查询示例做好准备。

一切准备就绪后, 请执行以下这个 Term Vector 查询示例:

```
http://localhost:8080/solr/techproducts/tv?q=includes:[*%20TO%20*]&rows=10&
indent=true&tv=true&tv.tf=true&tv.df=true&tv.positions=true&tv.offsets=true&tv.
payloads=true&tv.fl=includes
```

返回的查询结果集如下所示:

```
"termVectors": [
  "uniqueKeyFieldName", "id",
  "warnings", [
    "noPayloads", ["includes"]],
    "MA147LL/A", [
      "uniqueKey", "MA147LL/A",
      "includes", [
        "cable", [
          "tf", 1,
          "positions", [
            "position", 3],
          "offsets", [
            "start", 23,
            "end", 28],
          "df", 2],
```

...../ 其余部分省略

你可以为 Term Vector 组件设置请求参数来控制 Term Vector 组件返回的 Term 信息内容, Term Vector 组件支持接收以下请求参数, 具体请看表 11-15 中的详细解释。

表 11-15 Term Vector 组件参数表

参 数	描 述
tv	表示是否启用 TermsVector 组件, 默认值 false
tv.docIds	返回指定 DocID (不是 schema.xml 中配置的 uniqueKey) 的索引文档 Term Vector 信息, 多个 DocID 使用逗号分隔
tv.fl	表示返回指定域的 Term Vector 信息, 多个域名称使用逗号分割, 若此参数未指定, 则默认会以 fl 参数为准

(续)

参 数	描 述
tv.all	表示启用此参数下方的所有 boolean 参数
tv.df	表示是否返回 Document 中包含 Term 的频率, 这个计算开销很大, 你懂的
tv.offsets	表示是否返回 Term 的位置偏移量
tv.positions	表示是否返回 Term 的位置信息即 position 属性
tv.payloads	表示是否返回 Term 上负载的 Payload 信息
tv.tf	表示是否返回 Term 在每个索引文档中的出现频率
tv.tf_idf	表示是否计算每个 Term 的 TF/DF 值 (即 $TF * IDF$), 请注意, 这仅仅是简单的 TF 值乘以 IDF 值, 并不是经典的 TF-IDF 相似度计算。此参数需要 tv.tf 和 tv.df 两个参数也同时开启, 这个参数计算开销会非常大

11.12 Solr Query Elevation 组件

Solr 中 Query Elevation 组件允许你通过为给定的查询配置特定的 Top N 结果集, 并且不关心 Lucene 正常的打分机制。这通常也被称为赞助商查询, 这好比百度最赚钱的业务: 竞价排名。比如某个企业付费了, 我就优先将有关该企业的信息靠前显示, 而直接无视底层的打分机制。即人为的决定某些索引文档显示顺序。当你有这方面需求时, 你可能会需要使用 Query Elevation 组件。尽管此组件可以应用于任意 Query Parser (查询语法解析器), 但是将其与 DisMax 或者 eDisMax 结合使用会更有意义。此外, Query Elevation 组件还支持分布式查询。

默认 Query Elevation 查询组件并没有开启, 如果想要使用 Query Elevation 组件, 你需要显式的在 solrconfig.xml 中配置 Query Elevation 组件, 配置示例如下所示:

```
<searchComponent name="elevator" class="solr.QueryElevationComponent" >
  <str name="queryFieldType">string</str>
  <str name="config-file">elevate.xml</str>
  <!-- 此参数用于区分正常的结果集与编辑后的结果集 -->
  <str name="editorialMarkerFieldName">foo</str>
</searchComponent>

<requestHandler name="/elevate" class="solr.SearchHandler" startup="lazy">
  <lst name="defaults">
    <!-- 打印参数 -->
    <str name="echoParams">explicit</str>
  </lst>
  <arr name="last-components">
    <str>elevator</str>
  </arr>
</requestHandler>
```

表 11-16 列举了 Query Elevation 组件支持的所有配置参数。

表 11-16 Query Elevation 组件参数表

参 数	描 述
queryFieldType	指定对于用户输入的查询文本使用什么域类型进行分词处理
config-file	指定 elevate.xml 配置文件的加载路径, 此文件必须存放于 <instanceDir>/conf/ 或者 <instanceDir>/<dataDir> 目录下。<instanceDir> 表示你的 core 根目录, <dataDir> 表示 core 的数据目录。如果此文件存在于 conf 目录下, 那么它会在 Core 启动时被加载, 如果它存在于 data 目录下, 那么每个 IndexReader 创建时都会重新加载 elevate.xml
forceElevation	默认 Query Elevation 组件会使用 sort 参数来排序, 如果 forceElevation=true, 那么此组件会首先返回 elevate.xml 中定义的索引文档, 然后再按照 sort 参数对剩余的索引文档进行排序

elevate.xml 配置示例如下所示:

```
<elevate>
  <query text="foo bar">
    <doc id="1" />
    <doc id="2" />
    <doc id="3" />
  </query>

  <query text="ipod">
    <doc id="MA147LL/A" /><!-- 当用户输入 ipod 关键字进行搜索时, 将 id=MA147LL/A 的索引文档
放在首位显示 -->
    <doc id="IW-02" exclude="true" /><!-- 排除掉这个索引文档不返回显示, 即便它匹配了用户输
入的搜索关键字 -->
  </query>
</elevate>
```

query 用来定义对哪些查询返回的结果集进行重新定义, text 属性表示用户输入的搜索关键, 即当用户搜索时输入的这个关键字, 那么将应用下面定义的人为设置 TOP N 文档。比如上面的 <query text="foo bar"> 部分表示当用户输入 "foo bar" 关键字搜索时, 首先将 id=1,2,3 的索引文档放在前 3 位显示, 之后的按正常逻辑返回, 即人为的决定某些索引文档的显示顺序。你还可以在 <doc> 元素里通过指定 exclude="true" 来排除某些索引文档不返回显示, 即便它匹配了用户输入的搜索关键字, 比如该搜索结果是竞争对手的, 你可能希望屏蔽掉不显示。

以下是一个 Query Elevation 组件的简单查询示例, 如下所示:

```
http://localhost:8080/solr/techproducts/elevate?wt=json&indent=true&q=ipod&df=text&debugQuery=true&enableElevation=true&forceElevation=true
```

如上所示, 开启 Query Elevation 组件需要设置 enableElevation=true, 也可以直接将 enableElevation 参数配置在 Request Handler 的 defaults 属性部分, 比如:

```
<requestHandler name="/elevate" class="solr.SearchHandler" startup="lazy">
  <lst name="defaults">
    <str name="wt">json</str>
```

```

<bool name="indent">true</bool>
<!-- 配置默认启用 Query Elevation 组件 -->
<bool name="enableElevation">true</bool>
<str name="echoParams">explicit</str>
</lst>
<arr name="last-components">
<str>elevator</str>
</arr>
</requestHandler>

```

你可以在请求中添加 `forceElevation` 参数将其设置为 `true`，表示强制使用 `elevate.xml` 配置的人为文档加权规则。也可以在请求中添加参数 `exclusive=true`，表示告诉 Solr 只返回 `elevate.xml` 中定义的索引文档，其他索引文档不返回。你还可以为返回的查询结果集添加 `[elevated]` 和 `[excluded]` 这两个伪域，查询示例如下所示：

```

http://localhost:8080/solr/techproducts/elevate?q=ipod&df=text&fl=id,[elevated],
[excluded]&enableElevation=true&forceElevation=true

```

其中：

□ `[elevated]` 表示标识返回的每个索引文档是否是人为推荐的。

□ `[excluded]` 表示该文档应该排除掉你不希望返回给用户，前提是你查询时提供了 `markExcludes=true` 参数。

返回的结果可能是类似下面这样的：

```

{"id":"6H500F0",
  "[elevated]":true,
  "[excluded]":false},

```

前面我们都是直接在 `elevate.xml` 中定义哪些索引文档应该优先显示以及排除或者屏蔽个别索引文档，但是提前在 `elevate.xml` 中定义具有很大的局限性，因为我们的索引文档可能每天都在增长，人为推荐的规则也可能随时都在变化，那我们岂不是需要频繁的更新 `elevate.xml`，更新 `elevate.xml` 之后又得重新加载 Core，而重新加载 Core 是一个很昂贵的操作。因此采用 `elevate.xml` 的方式对于大部分使用场景来说，并不适用。此时你可以通过设置 `elevateIds` 和 `excludeIds` 参数来动态的指定对哪些索引文档进行人为推荐或者屏蔽。请看以下的查询示例：

```

http://localhost:8080/solr/techproducts/elevate?q=cable&df=text&excludeIds=IW-02&elevateIds=3007WFP,9885A004

```

上面的查询示例表示对文档 id 为 3007WFP,9885A004 进行人为推荐置顶显示，而对于文档 id 为 IW-02 的索引文档进行屏蔽不显示。如果为某个查询启用了 `elevateIds` 或者 `excludeIds` 参数其中任意一个，将会无视 `elevate.xml` 中针对该查询的一切配置，注意，仅仅是覆盖该查询的配置，即 `elevate.xml` 中 `<query text="cable">` 这部分的配置失效，而不是整个 `elevate.xml` 配置失效。

不管是 q 参数的主查询还是 fq 参数的 Filter Query, elevate.xml 中定义的索引文档都必须符合这两个参数的查询条件, 否则即便你将它们置顶显示也无济于事。换句话说, Query Elevation 组件只适用于符合查询条件的索引文档, 人为的“提拔”个别索引文档置顶显示或者人为的剔除几个索引文档进行屏蔽, 但是你不能人为的向 elevate.xml 中添加不符合查询条件的索引文档, 因为那样无法生效。

11.13 Solr Result Clustering 组件

Solr 中的 Result Clustering 组件会尝试自动发现相关搜索结果集的分组并为这些分组指定人类可读的标签。默认在 Solr 中, Clustering 这种自动分组聚类的算法会被应用于每个单个查询请求, 被称为: 在线聚类。然而 Solr 包含一种全文聚类 (也叫离线聚类) 扩展, 我们主要关注在线聚类。对于给定的查询能够自动聚类可以视为一个动态 Facet (维度), 当域的域值无法预知, 常规的 Facet 就无法开始, 或者当查询本质上就是探索性的即如何定义 Facet (维度) 尚不明确, 此时使用 Clustering 来自动聚类分组会很有用。关于 Result Clustering, 这里有个单独的框架: Carrot2, 它的官网地址为:

<http://project.carrot2.org/>

Carrot2 支持从各种搜索引擎 (YahooAPI、GoogleAPI、MSN Search API、eTools Meta Search、Alexa Web Search、PubMed、OpenSearch、Lucene index、SOLR) 获取搜索结果。

Solr 中的 Facet 不能生成一组类似的分组, 尽管两者的技术目的是相似的。Result Clustering 和 Result Group 也很相似, 不仅仅只是让你看到 Top N 条命中的结果集, 更多的是帮你更深层次的查阅浏览查询结果集。

传递给 Result Clustering 组件的每个索引文档由以下几个逻辑部分组成:

❑ 一个唯一的标识符

❑ 原始 URL

❑ title (标题)

❑ 主要内容即 content

❑ title 和 content 的语言代号

唯一标识符部分是强制需要的, title 和 content 域至少提供一个, 其他的都是可选的。逻辑文档部分必须要映射到特定的 schema。

下面我们开始尝试在 Solr 中使用 Result Clustering, 首先你需要在 solrconfig.xml 中 ClusteringComponent 组件, 配置示例如下所示:

```
<searchComponent name="clustering"
  enable="${solr.clustering.enabled:false}"
  class="solr.clustering.ClusteringComponent" >
  <lst name="engine">
```

```

<str name="name">lingo</str>
<str name="carrot.algorithm">
    org.carrot2.clustering.lingo.LingoClusteringAlgorithm
</str>
<!-- 特定算法需要加载的配置文件存放路径, 默认值 conf/clustering/carrot2/-->
<str name="carrot.resourcesDir">clustering/carrot2</str>
</lst>
<lst name="engine">
<str name="name">stc</str>
<!-- Solr 提供的可选的算法有:
org.carrot2.clustering.lingo.LingoClusteringAlgorithm
org.carrot2.clustering.stc.STCClusteringAlgorithm
org.carrot2.clustering.kmeans.BisectingKMeansClusteringAlgorithm
-->
<str name="carrot.algorithm">
    org.carrot2.clustering.stc.STCClusteringAlgorithm
</str>
</lst>
<lst name="engine">
<str name="name">kmeans</str>
<str name="carrot.algorithm">
    org.carrot2.clustering.kmeans.BisectingKMeansClusteringAlgorithm
</str>
</lst>
</searchComponent>
<requestHandler name="/clustering"
    startup="lazy"
    enable="{solr.clustering.enabled:false}"
    class="solr.SearchHandler">
<lst name="defaults">
<str name="wt">json</str>
<bool name="indent">true</bool>
<!-- 是否开启 Result Clustering -->
<bool name="clustering">true</bool>
<bool name="clustering.results">true</bool>
<!-- 对于 carrot 需要的逻辑 "title" (可选) -->
<str name="carrot.title">name</str>
<!-- 对于 carrot 需要的逻辑 "url" (可选) -->
<str name="carrot.url">id</str>
<!-- 对于 carrot 需要的逻辑 "content" (可选) -->
<str name="carrot.snippet">features</str>
<!-- 是否对逻辑 "title/content" 开启高亮器 (可选) -->
<bool name="carrot.produceSummary">true</bool>
<!-- Label 标签的最大个数 (可选) -->
<!--<int name="carrot.numDescriptions">5</int>-->
<!-- 是否是否生成子 Cluster (可选) -->
<bool name="carrot.outputSubClusters">>false</bool>
<str name="defType">edismax</str>
<str name="qf">
    text^0.5 features^1.0 name^1.2 sku^1.5 id^10.0 manu^1.1 cat^1.4

```

```

</str>
<str name="q.alt">*:*</str>
<str name="rows">10</str>
<str name="fl">*,score</str>
</lst>
<arr name="last-components">
<str>clustering</str>
</arr>
</requestHandler>

```

然后 `core.properties` 配置文件中设置 `solr.clustering.enabled=true` 即启用 clustering, 配置示例如下所示:

```

name=techproducts
config=solrconfig.xml
schema=schema.xml
dataDir=data
solr.clustering.enabled=true

```

然后你需要在当前 `core` 的 `\conf\clustering\carrot2` 目录下添加如下 3 个配置文件:

```
kmeans-attributes.xml、lingo-attributes.xml、stc-attributes.xml
```

然后你需要将 `solr-5.3.1\contrib\clustering\lib` 目录下的 jar 以及 `solr-clustering-5.3.1.jar` 导入到当前 `core` 的 `lib` 目录下, 然后重新加载 `Core` 即可。更详细的配置请查看随书源码提供的 "techproducts" `Core` 相关文件。

Result Clustering 配置完成之后, 请打开浏览器执行下面这个查询示例:

```
http://localhost:8080/solr/techproducts/clustering?q=*:*&rows=100
```

返回的结果集部分如下所示:

```

"clusters": [{
  "labels": ["DDR"],
  "score": 1.46243794606717,
  "docs": ["TWINX2048-3200PRO",
    "VS1GB400C3",
    "VDBDB1A16"]},
{
  "labels": ["iPod"],
  "score": 5.571285181190992,
  "docs": ["F8V7067-APL-KIT",
    "IW-02",
    "MA147LL/A"]},
.....// 省略部分

```

从以上返回的查询结果集可以发现, 我们只是查询所有索引文档, 并没有对任何域进行 Facet, 也没有执行分组查询, 但是返回的索引文档还会被自动划分为 ["DDR"]、["iPod"]、["Retail"]、["Other Topics"] 这 4 组。这就是 **Result Clustering** 的作用。

11.14 本章总结

在本章中，我们系统性的学习了 Solr 中的函数查询以及如何自定义 Function，并学习了 Solr 中强大的地理空间查询，实现类似“离当前位置方圆多少千米以内的酒店”之类的查询。然后学习 Solr 的 Pivot Facet 多维度嵌套查询，由于 Solr 中的 Pivot Facet 具有一定的局限性，于是 Solr 重写了 Facet 查询模块，设计了全新的 JSON Facet API，其中 Subfacet 就是采用 JSON Facet API 实现，而 Subfacet 强大的数据统计功能又是借助于 Facet Function 来完成。同时 Subfacet 支持任意深度的嵌套，恰好弥补了传统 Pivot Facet 的不足。紧接着我们又陆续学习 Solr 中提供的各种查询组件，比如 Terms 组件、Term Vector 组件、Stats 数据统计组件、Query Elevation 竞价排名组件、Result Clustering 自动聚类分组组件等等。通常本章的学习，你完全可以构造出更复杂的查询来满足项目需求，通过 JSON Facet API 你能完成大部分的数据查询统计工作，并使用一些可视化组件比如矩状图、饼状图等形式将统计数据形象的展现出来。下一章我们将继续学习 Solr 的 Join 查询以及如何进一步提高查询返回的结果集的相关度。

Solr 查询进阶篇

通过第 12 章，你将可以学习到以下内容：

- 掌握如何在 Solr 中使用游标实现高效的深度分页查询；
- 掌握如何在 Solr 中实现对查询返回的查询结果集进行自定义排序；
- 掌握如何使用 Solr 中的 Join 实现跨索引文档跨 Core 查询；
- 掌握如何使用相关性权重来提高查询返回结果集的相关性；
- 掌握 Solr 中的 NRT（近实时）查询；
- 掌握 Solr 中的 Real-time Get 查询来实时获取最新版本的索引文档；
- 掌握如何使用 Ranking Query 对 Top N 索引文档重新评分；
- 掌握如何使用 Solr 中的 MoreLikeThis 组件；
- 掌握在 Solr 中如何自定义 Query Parser。

12.1 Solr 深度分页

在 Solr 中，默认分页查询需要使用 `start` 和 `rows` 参数，分页查询的性能调优可以通过启用 `queryResultCache` 和根据你实际分页的每页显示大小调整 `queryResultWindowSize` 参数来实现。

一般情况下，默认的分页查询结合 `QueryResultCache`（查询结果集缓存）运行不会有什么大问题，但是极端情况下，假如希望查询第 100 000 页且每页显示 10 条，意味着 Solr 提取前 $100\,000 \times 10 = 1\,000\,000$ 条数据，并将这 1 000 000 条数据缓存在内存中，然后在内存中对其排序，最后返回最后 10 条即用户想要的第 100 000 页数据。首先在内存中缓

存 1 000 000 条数据需要占用很多内存，并且在内存中对 1 000 000 条数据进行排序也很耗 CPU，因此，默认的分页方式只适合于查询 Top N 页数据，翻页越到后面，分页查询性能越差。当然对于大部分使用场景来说，用户也不太可能那么无聊的翻到第 100 页，但是不排除用户可以直接修改请求 URL 直接跳到第 100 页恶意的攻击你的搜索服务器。

默认分页请求的 `start` 参数表示的是已排序的整个结果集中的绝对索引位置偏移量。如果在分页查询的同时有索引文档被修改了（比如添加或删除了索引文档），将会影响已排序的整个结果集中每个索引文档的索引位置，这使得一个索引文档可能会在前后两次分页中出现或者被跳过。举例说明，假如我们有如下这样的 26 个索引文档：

id	name
1	A
2	B
...	// 省略
26	Z

假设用户执行如下操作：

1) 用户查询第一页数据：`q=*&rows=5&start=0&sort=name asc`

此时会返回 document 1 ~ 5。

2) 假设此时 document 3 被其他用户删除，同时用户请求第 2 页数据：`q=*&rows=5&start=5&sort=name asc`

此时 document 7-11 被返回，而 document 6 被跳过，因为此时它是第 5 个 document 属于第 1 页，也就是说，你如果再查询第 1 页，返回的 document 为 1, 2, 4, 5, 6。

3) 此时新添加了 id 为 90,91,92 这 3 个 document，同时用户请求第 3 页数据：`q=*&rows=5&start=10&sort=name asc`

此时返回 document 9-13，你会发现 document 9,10,11 在第 2 页和第 3 页都返回了。

在一般情况下，索引数据更新对默认分页查询的影响并不会影响用户的使用体验，这发生的一般不会太频繁，再个数据本身也确实是在不断变化，因此数据的排序发生改变用户也不会感到诧异。

在某些情况下，你可能希望一页返回大量数据，或者你想查询第 100 000 页数据，这样你的 `start` 或 `rows` 参数需要设置的很大，这在默认的分页查询模式下，性能是很差的，即便你开启了 `QueryResultCache`，这种查询会大量消耗你的内存和 CPU，仅仅只是为了得到一页数据付出的代价确实不划算。

为此 Solr 提供了一种全新的分页方式，提出了 "Cursor" 游标的概念来解决上述默认分页查询的性能问题。Solr 中的游标是一个逻辑概念，它不会在服务器上存储任何信息，而是返回一个下一页数据起始位置 "Mark" 标记值给用户，该标记表示着当前分页起始位置在查询匹配的整个索引结果集中的绝对索引位置。简而言之，游标中包含了分页查询结果集的偏移量。因此，Solr 不再需要每次从头开始遍历结果直到我们想要的记录。用户每次分页提供当前 Mark 标记值即可。游标的设计可以大幅提升深翻页的性能，但是是以消耗内存

为代价的。

想要在 Solr 中使用游标，你需要指定一个 `cursorMark` 参数比如：`cursorMark=*`，你可以理解为它跟 `start=0` 类似。然后此时 Solr 除了会返回一个 Top N 的结果集，同时还附带返回一个 `nextCursorMark` 值，`nextCursorMark` 表示游标下一次遍历的起始位置即下一次分页从 `nextCursorMark` 位置开始返回。`nextCursorMark` 值是查询匹配结果集中的数据索引位置的编码值，每一次分页查询都需要带上 `cursorMark` 参数即 `cursorMark=nextCursorMark` 值（第一页除外），你可以重复这个过程，直到 Solr 返回的 `nextCursorMark=cursorMark`，那么就表明此时已经没有下一页了。

但是 Solr 中使用游标实现的深度分页有几点比较重要的限制：

- ❑ `cursorMark` 和 `start` 参数是互斥的，你不能同时指定这两个参数，或者你同时指定两个参数可以，但是此时 `start` 参数必须等于默认值零。
- ❑ `sort` 语句必须包含唯一主键域，如果 `id` 是你的主键域，那么 `sort` 参数可以像这样设置：`sort=id asc,name asc`。但是你不能设置 `sort=name desc`。

游标标记是根据结果集中的每个索引文档的排序值进行计算而来的，这意味着如果两个索引文档的排序值相同，那么它们生成的游标标记值也相同，这也是为什么上面要求 `sort` 语句必须包含唯一主键域的原因，因为一般唯一主键是不可能相同的，从而保证了生成的 `Mark` 标记值不会相同。

Solr 中使用游标深度分页查询的示例如下所示：

```
// 查询第一页，此时必须 cursorMark=*
http://localhost:8080/solr/techproducts/select?
    sort=name desc, id asc&q=name:ipod*&fl=id,name&
    rows=1&wt=json&indent=true&cursorMark=*
```

返回的结果集如下所示：

```
"response":{"numFound":3,"start":0,"docs":[{"
    "id":"F8V7067-APL-KIT",
    "name":"Belkin Mobile Power Cord for iPod w/ Dock"}]
},
"nextCursorMark":"AoImYmVsa2luL0Y4VjcwNjctQVBMLUtJVA=="}
```

此时结果集里会返回 `nextCursorMark` 值，那么接下来的分页查询需要每次通过 `cursorMark` 参数将上一次分页查询返回的 `nextCursorMark` 值再传递给 Solr。因此此时请求第 2 页查询的请求 URL 如下所示：

```
http://localhost:8080/solr/techproducts/select?
    sort=name desc, id asc&q=name:ipod*&fl=id,name&
    rows=1&wt=json&indent=true&cursorMark=AoImYmVsa2luL0Y4VjcwNjctQVBMLUtJVA==
```

注意这里的 `cursorMark` 参数值，它需要与上一次分页查询结果集里返回的 `nextCursorMark` 属性值保持一致，直到返回的 `nextCursorMark` 等于当前的 `cursorMark`，也就表明分页到底了。

正由于每次请求下一页都需要上一页查询返回的 `nextCursorMark` 游标标记值，这也就意味着，在使用 Solr 游标实现深度分页查询时，你无法实现跳到任意指定页（第一页除外）的分页查询需求，因为在这种分页模式下，你只能一页一页地翻下去，就好比链表结构遍历，你也只能从头节点一直 `next` 下去直到尾节点。

接下来我们考虑下索引数据更新会对使用游标实现的深度分页查询带来什么影响？这里为了表述更形象，以排队报数为例进行讲解说明，假如有 20 个人站一排依次给他们编号：1, 2, 3, 4, ..., 19, 20，并按照编号从小到大排序。第一次先从 1 号开始报数，每次报数报 5 个，此时会停在 5 号上即 1, 2, 3, 4, [5], 6, ..., 19, 20。假如此时 3 号从队伍中退出，进行下一轮报数，会从 6 号开始报数，报到 10 号为止。3 号的退出并不影响报数过程。因为每次会从上一次停留的位置往后数 5 个，但是如果此时退出的是 6 号，那么就会有影响，此时会从 7 号开始直到 11 号。假如此时将 6 号的编号改成 3 号，那么此时队伍变成：1, 2, 3, 4, 5, 7, 8, ..., 19, 20。还是从 5 号的下一位开始报数，即此时是从 7 号开始报数直到 11 号。类比到我们 Solr 中的游标，内部机制大致是相似的，也就是说索引数据的更新几乎不会影响 Solr 中采用游标实现的深度分页。但是如果结果集的排序域的域值改变了，可能会影响深度分页查询返回的结果集。比如我们上面的查询示例中按照 `name` 和 `id` 域排序，假如索引的 `name` 域的域值被更新了，可能会影响它在查询结果集中的绝对位置，就好比我们所举的报数例子中某个人的编号值被改变了。它的绝对位置改变了，就会影响游标标记值的相对位置，从而影响分页最终返回的结果集。

12.2 Solr 自定义排序

Solr 中实现排序，只需要添加 `sort` 参数即可，比如 `sort=price asc` 表示按照价格从低到高排序。有时候你希望按照某两个域的值经过一定的计算，根据计算后得到的值进行排序，你可以通过自定义函数的方式实现。关于如何在 Solr 中自定义函数，请翻到第 11 章的第 1 节的《Solr 函数查询》部分进行回顾。

考虑这样一个场景，比如你 MySQL 数据库里有个 `blog` 表，表结构如图 12-1 所示。

Field	Type	Null	Key	Default	Extra
<code>id</code>	<code>bigint(20)</code>	NO	PRI	NULL	<code>auto_increment</code>
<code>title</code>	<code>varchar(200)</code>	YES		NULL	
<code>thumbs_up</code>	<code>bigint(20)</code>	YES		NULL	
<code>thumbs_down</code>	<code>bigint(20)</code>	YES		NULL	
<code>total_votes</code>	<code>bigint(20)</code>	YES		NULL	

图 12-1 weibo 测试表表结构

其中 `thumbs_up` 表示点赞人数，`thumbs_down` 表示点踩人数，`total_votes` 表示总参与人数，测试数据如图 12-2 所示。

id	title	thumbs_up	thumbs_down	total_votes
1	【一代神机说再见！iPhone 4退出历史舞台】	100000	28	100028
2	梁朝伟代言小米系列	9999999	3	10000002
3	iOS 10.0.3 更新推送，貌似只针对iPhone 7和iPhone 7 Plus	666	9	675

图 12-2 weibo 表测试数据

假设想要如下计算评分：

点赞率 = $\text{Math.round}(\text{thumbs_up} * 100 / \text{total_votes})$

点踩率 = $\text{Math.round}(\text{thumbs_down} * 100 / \text{total_votes})$

评分 = 点赞率 - 点踩率 + 100

在 Solr 里，根据上面的评分计算公式最终计算得到的评分从高到低对每条微博进行排序，此时应该如何实现呢？解决此类问题的大致思路就是：

1) 先自定义一个 FieldType (域类型)，该类的 getSortField() 方法需要使用自定义的 FieldComparatorSource 类返回一个自定义的 SortField。

2) 在 schema.xml 中配置这个自定义的 FieldType。

3) 创建自定义的 FieldComparatorSource 并在它的新 newComparator() 方法里返回一个自定义的 FieldComparator。这个自定义的 FieldComparator 需要通过 JDBC 方式从数据库加载数据并按照我们预定的计算公式计算评分。

4) 最后就能在查询时通过 &sort=rank desc 来按照自定义的评分方式进行排序了。

下面请随我一起实现这个需求，首先将 MySQL 里的 weibo 这张表里的数据导入到 Solr 中建立索引。其次需要确定那些字段需要在 Solr 的 schema.xml 中定义域呢。因为 thumbs_up, thumbs_down, total_votes 这 3 个字段的数据可能每分每秒都在更新，如果索引到 Solr 中，也就意味着需要频繁的更新 Solr 数据与之保持同步，这显然不合理，频繁更新 Solr 索引数据性能也不好，同时索引更新还会影响 Solr 的实时搜索功能。因此这里我们只索引 id 和 title 域，schema.xml 配置示例如下所示：

```
<field name="id" type="integer" indexed="true" stored="true" />
<field name="title" type="text_path" indexed="true" stored="true"/>
```

更详细的配置请查阅随书源码中提供的 "weibo" Core，同时根据 IndexWeibo 测试类将数据导入到我们创建的 "weibo" Core。

第一步，创建 RankFieldType，以下是部分关键代码，完整的源码文件请从随书源码里查找。

```
public class RankFieldType extends FieldType {
    .../
    @Override
    public SortField getSortField(SchemaField field, boolean top) {
        return new SortField(field.getName(),
```

```

        new RankFieldComparatorSource(), top);
    }
}

```

创建一个 SortField 需要实现自己的 FieldComparatorSource, 因此我们自定义了 RankFieldComparatorSource, 而 FieldComparatorSource 又需要返回一个 FieldComparator 用来比较大小, 排序的时候会根据这个类决定域值在结果集中的排列顺序, 下面是 RankFieldComparatorSource 和 FieldComparator 的部分实现代码:

```

public class RankFieldComparatorSource extends FieldComparatorSource {
    public FieldComparator newComparator(String fieldname, int numHits,
        int sortPos, boolean reversed) throws IOException {
        return new RankFieldComparator(numHits);
    }
}

public static final class RankFieldComparator extends FieldComparator<Integer>
implements LeafFieldComparator {
    private final int[] docIDs;
    private int docBase;
    private int bottom;
    private int topValue;
    RankFieldComparator(int numHits) {
        docIDs = new int[numHits];
    }
    public int compare(int slot1, int slot2) {
        return getRank(docIDs[slot1]) - getRank(docIDs[slot2]);
    }
    private Integer getRank(int docId) {
        return RankUpdateListener.getRank(docId);
    }
}

```

通过 RankUpdateListener 监听器监听 IndexSearcher 创建事件, 当 IndexSearcher 被创建时, 通过 JDBC 连接数据库查询数据, 按照自定义的计算公式计算评分, 然后返回给 RankFieldComparator, 这样 RankFieldComparator 才能根据在 compare 方法中根据评分去比较大小决定索引文档的顺序。最后你需要自定义 RankExtractComponent 查询组件, 并在 Request Handler 中应用该查询组件, 从而使其生效。关于这部分的完整实现, 请直接查阅随书源码, 因为源码太长, 不便一一贴出。

RankUpdateListener 创建好之后, 需要在 solrconfig.xml 中注册这个监听器, 配置示例如下所示:

```

<listener event="firstSearcher"
    class="com.yida.solr.book.examples.ch12.customsort.listener.
RankUpdateListener"/>
<listener event="newSearcher"      class="com.yida.solr.book.examples.ch12.
customsort.listener.RankUpdateListener"/>

```


同时，还需要在 `solrconfig.xml` 中注册我们自定义的 `RankExtractComponent` 查询组件，配置示例如下所示：

```
<searchComponent name="rank-extract"
  class="com.yida.solr.book.examples.ch12.customsort.component.
    RankExtractComponent"/>
<requestHandler name="/select" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">none</str>
    <str name="wt">json</str>
    <str name="indent">true</str>
    <str name="df">text</str>
  </lst>
  <arr name="last-components">
    <str>rank-extract</str>
  </arr>
</requestHandler>
```

在自定义 `RankExtractComponent` 查询组件中，我们通过 `RankUpdateListener` 监听器获取点赞数、点踩数以及最终计算得到的文档评分 `rank`，并将它们作为 3 个域添加到查询结果集中的每个索引文档中，注意：是往查询时返回的结果集中的每个索引文档追加域，我们在创建索引时并没有创建这 3 个域，但是创建域前提是该域在 `schema.xml` 中存在，因此还需要在 `schema.xml` 中额外追加 3 个域，配置示例如下所示：

```
<field name="thumbs_up" type="integer" indexed="true" stored="true" />
<field name="thumbs_down" type="integer" indexed="true" stored="true" />
<field name="rank" type="integer" indexed="true" stored="true" />
```

此时你可以重新加载 `Core`，然后执行如下查询示例进行测试：

```
http://localhost:8080/solr/weibo/select?q=*&fl=id,title,rank&sort=rank desc
```

最后返回的查询结果如下所示：

```
"response":{"numFound":3,"start":0,"maxScore":"NaN","docs":[
  {
    "id":"1",
    "title":"【一代神机说再见！ iPhone 4 退出历史舞台】",
    "rank":200},
  {
    "id":"2",
    "title":"梁朝伟代言小米系列",
    "rank":200},
  {
    "id":"3",
    "title":"iOS 10.0.3 更新推送，貌似只针对 iPhone 7 和 iPhone 7 Plus",
    "rank":198}]}
```

正如你看到的那样，结果集中的每个索引文档都返回了一个 rank 域，然而我们在创建索引时并没有显式的添加 rank 域，rank 域是我们自定义的 RankExtractComponent 查询组件在查询结果集返回之前，从数据库加载数据计算出 rank 评分，然后临时添加了 3 个域，但这 3 个域并不存在于硬盘的索引数据中。我们的示例中是通过 JDBC 去加载关系型数据库，当然你可以从任意外部数据源加载数据，然后动态添加域，比如从 HBase 加载数据，从外部的 txt、xml、excel、pdf 等文件中加载数据。不过从外部文件加载数据，此时你可以使用 Solr 中内置的 ExternalFileField 域来实现。关于 ExternalFileField 的详细介绍在 3.3.5 章节已经介绍过了，请按照说明在 schema.xml 中配置 ExternalFileField 域，同时，在 solrconfig.xml 中的 <<requestHandler> 元素附近配置一个监听器，配置示例如下所示：

```
<listener event="newSearcher"  
class="org.apache.solr.schema.ExternalFileFieldReloader"/>  
<listener event="firstSearcher"  
class="org.apache.solr.schema.ExternalFileFieldReloader"/>  
<requestHandler name="/select" class="solr.SearchHandler">  
<lst name="defaults">  
.....// 其他省略
```

你会发现，通过这种方式，我们不仅能实现自定义排序，还能解决数据同步问题，即那些更新很频繁的数据你可以不用写入到 Solr 索引中，而是在查询时动态的从外部数据源加载然后通过动态添加域的方式返回给用户。这样就不存在数据同步问题，这种方式虽然解决了数据同步问题，但是也有一定的局限性。首先，我们只是在查询时动态添加了该域，但是该域在索引数据中并不存在，也就意味着你不能根据这些查询时动态添加的域（即我们示例中的 rank、thumbs_up、thumbs_down 这 3 个域）进行查询，即当你通过 q=rank:[100 TO 200] 进行查询时，是查询不到数据的，因为索引数据中 rank 域没有数据。第二点，我们的数据每次都是通过 JDBC 去关系型数据临时加载的，这会有一定的性能损耗，比如网络 IO、SQL 执行性能这都会影响我们 Solr 查询最终的性能，当然你可以为 JDBC 查询加缓存进行缓解，比如先判断 Redis 缓存中存不存在，不存在再去 JDBC。当然也有可能是其他方式比如磁盘 IO 加载文件的方式，此时若磁盘 IO 有阻塞会导致我们的 Solr 查询出现阻塞，这也是你需要考虑到的潜在问题。此外，Solr 中的自定义排序还可以通过自定义函数的方式实现，关于如何在 Solr 中自定义函数，请翻到 11.1.6 章节进行回顾学习。至此，Solr 中自定义排序就介绍完毕了。

12.3 Solr Join 查询

Solr 中的索引文档一般建议扁平化、非关系化。所谓扁平化即每个索引文档包含的域类型和域个数可以不同，而非关系型化则表示每个 Document 之间没有任何关联，是相互独立的，不像关系型数据库中两张表之间可以通过外键建立联系。

#user		#country	
id	bigint	id	bigint
user_name	varchar(20)	country_name	varchar(20)
male	bit	country_code	varchar(10)
birth			
country	bigint		

图 12-3 user 和 country 表示例

如图 12-3 所示，为两张表，如果想要对这样两张表建立索引，一般建议是将 country_name 和 country_code 这两个字段冗余到 user 表里，然后统一创建一个 "user" Core 即可。但这并不是恒久不变的定理，特殊情况还是可以按照关系型数据库那样通过引用外键的方式创建两个单独的 Core。比如当外键关联的表里面包含的字段数目太多时，全部冗余到一个 Core 里可能不太好管理，你可以按照关系型数据库里建表的思想来创建你的 Core，然后利用 Solr 提供的 Join 查询能力来实现类似 SQL 里的嵌套子查询功能。Solr 中的 Join 查询语法类似如下示例：

```
/select?
fl=RETURN_FIELD_1, RETURN_FIELD_2&
q={!join from=FROM_FIELD
to=TO_FIELD)CONSTRAINT_FIELD:CONSTRAINT_VALUE
```

而 SQL 里的嵌套子查询语法类似如下示例：

```
Select RETURN_FIELD_1, RETURN_FIELD_2 FROM join-data
WHERE TO_FIELD IN (
SELECT FROM_FIELD from join-data
WHERE CONSTRAINT_FIELD= 'CONSTRAINT_VALUE'
)
```

Solr 中的 Join 并不是真正意义上的 Join 查询，因为子查询返回的结果集并不能随着主查询一起，它只能用于限制主查询返回的结果集。即便如此，通过另一个查询来约束一个查询返回的结果集在某些时间很有用。

那么什么情况下我们需要使用 Solr 中的 Join 查询功能呢？假如有一个主索引文档（比如说饭店 restaurant），需要记录哪个饭店信息被什么用户访问了（比如记录单击访问次数），但是用户的访问行为是随时都有可能发生，也就是说你 restaurant 的索引数据中表示访问次数的域需要在每次用户单击访问之后频繁的更新，且不谈索引数据频繁更新带来的性能问题，频繁更新本身就是一件很痛苦的事情。有了 Solr Join 查询的支持，你完全可以将它们分成两个 Core 进行存储。Solr 要求 Join 查询涉及的索引文档必须属于同一个 Solr Server，同时也意味着 Solr 支持跨 Core 的 Join 查询。

12.3.1 跨 Core Join

仍然以我们上面的饭店为例，将饭店和用户的单击行为单独成两个 Core 进行存储，这样任意一个 Core 的数据更新不会影响到另一个 Core。Solr 中的 Join Query Parser 支持

fromIndex 和 toIndex 两个参数来启用跨 Core 的 Join 查询功能。假设下面是 restaurant(饭店) 和用户单击访问行为两个 Core 的索引文档的 Schema:

Restaurant core 的 schema.xml:

```
<field name="id" indexed="true" stored="true" />
<field name="restaurantname" indexed="true" stored="true" />
<field name="description" indexed="true" stored="false" />
```

useraction core 的 schema.xml:

```
<field name="id" type="string" indexed="true" stored="true" />
<field name="userid" type="string" indexed="true" stored="true" />
<field name="restaurantid" type="string" indexed="true" stored="true" />
<field name="actiontype" type="string" indexed="true" stored="false" />
<field name="actiondate" indexed="true" stored="false" />
```

为了便于本章节后续的查询演示, 请根据随书源码提供的 Core 配置文件以及导入数据测试类创建好 join_restaurants 和 join_useractions 这两个 Core 请导入测试数据。测试数据导入成功之后, 我们可以执行下面这个跨 Core Join 查询, 示例如下所示:

```
http://localhost:8080/solr/join_restaurants/select?
fl=restaurantname,text&q="Indian"&
fq={!join fromIndex=join_useractions
toIndex=join_restaurants from=restaurantid to=id}userid:user123 AND
actiontype:clicked
AND actiondate:[NOW-14DAYS TO *]
```

以上的查询示例表示返回 user123 用户最近 2 周单击访问 Indian 饭店的索引文档, 其中 fq 部分是在 "join_useractions" Core 上执行一个子查询, 然后利用子查询返回的结果集来约束在 "join_restaurants" Core 上的 q="Indian" 主查询。由于有两个查询, 因此会返回两个结果集, 两者通过一个外键进行 Join 连接。fromIndex 参数表示被连接的 Core, toIndex 表示主 Core, 主 Core 由请求 URL 前缀部分的 http://localhost:8080/solr/join_restaurants 这里指定的 Core 决定。from 和 to 参数用于进行主外键关联, from 参数表示外键。

Solr 中的 Join 查询是通过 Join Query Parser 实现的, 可以通过指定多个 fq 参数, 并在多个 fq 参数上使用 Join Query Parser, 从而能够实现在单个查询请求中执行多个 Join 查询。此外, 如果想要为每个 Join Query 使用 Nested Query (嵌套查询), 你可以使用 Boolean 连接符将多个 Join Query 连接起来。因为在嵌套查询中每个 Query Parser 独立执行的, 就好比主查询中通过 AND 或者 OR 连接的两个 Term。

尽管你不能将子查询返回的结果集中的索引文档包含的域进行返回, 但是可以根据子查询匹配的索引文档中的值来限制主查询返回的结果集。当索引文档中的某些域的值需要频繁的更新时, 可以考虑单独成两个 Core, 那么 Join 查询就可以派上用场了。

除了可以使用 Join Query Parser 实现 Join 查询, Solr 中还提供了很多高级 Query Parser 可以用于 Join 查询, 比如 Block Join Children Query Parser({!child of=...}) 以及 Block Join Parent

Query Parser(!parent of=...))。这两种 Join 查询实现方式能够使你的 Join 查询性能更好，但是这种方式需要你在索引的时候为索引文档定义父子关系。当你发现使用 Join Query Parser 实现 Join 查询性能很差时，可以考虑下采用 Block Join 这种 Join 查询性能更好的方式来实现。但是需要注意的是，在分布式查询模式下，不能跨多个节点实现跨 Core 查询。

12.3.2 跨 Document Join

不仅仅可以跨 Core Join，还可以实现跨 Document Join。假设数据库中有商品表 goods 和商品分类表 category，你想将它们导入到 Solr 并索引，此时可以按照上一章节的方式分成 goods 和 category 两个 Core 单独进行存储，然后使用跨 Core Join 查询来联合两个 Core 进行查询，你也可以这样建立你的 Core，将两者合并到一个 Core 里进行索引，即创建一个 "goods" Core，schema.xml 按照如下示例进行设计：

```
<field name="id" type="string" indexed="true" stored="true" />
<field name="goods_name" type="string" indexed="true" stored="true" />
<field name="category" type="string" indexed="true" stored="true" />
<field name="category_id" type="string" indexed="true" stored="true" />
```

其中 goods_name 域表示商品名称，category 域表示商品分类名称，id 域在这里即可以表示商品 id，也可以表示商品分类 id，因为这里我们将商品和商品分类融合到一个 Core 里了，但是融合到一个 Core 里之后，添加索引时商品的 ID 和商品分类的 ID 不要重复。category_id 域表示商品对应的商品 id，关联到商品分类 Document 的 id，即它是一个类似关系型数据库表中的外键，当 Document 的 category_id 域存在，则表明当前 Document 表示的是商品（因为只有商品才需要通过 category_id 域去关联商品分类 Document 的 id 域），当 Document 的 category_id 域不存在，则表明这是一个商品分类 Document。

请按照随书源码中提供的 Core 配置文件和测试数据导入类创建好 "goods" Core 并导入测试数据。假设此刻你已经完成了准备工作，下面请执行以下的查询示例，如下所示：

```
// 查询商品 "iPhone 7" 的商品分类
http://localhost:8080/solr/goods/select?q={!join from=category_id to=id}goods_
name:"iPhone 7"
```

类似于 SQL: select * from goods where id in(select category_id from goods where goods_name='iPhone 7')

```
// 查询商品分类 "Book" 下的所有商品
http://localhost:8080/solr/goods/select?q={!join from=id to=category_id}
category:"Book"
```

类似于 SQL: select * from goods where id in(select id from goods where category='Book')

```
// 查询商品分类 "手机" 下的名称为 "iPhone 7" 的商品
http://localhost:8080/solr/goods/select?q=goods_name:"iPhone 7"&fl=*,score&
```



```
sort=score desc&fq={!join from=id to=category_id}category:手机
```

这里使用了 fq 参数来进行子查询过滤, 因为 fq 参数构造的 Filter Query 会利用 Filter 缓存, 所以查询性能会较高。从上面的查询示例返回的查询结果集来看, 你不难发现, 跨 Document 的 Join 查询每次也只能返回某一种 Document, 即不能在商品 Document 中返回商品分类 Document 的 category 域, 并且不能在商品分类 Document 中返回商品 Document 的 goods_name 域。只能实现类似 SQL 里的 "select * from A where id (select s_id from B where xxx=?) " 这样的查询, 而不能实现类似 SQL 里的 left Join 或者 right join。此外跨 Document 的 Join 查询还有一点不足的地方就是查询返回的索引文档的评分都是相同的。即子查询匹配文档的评分不能用于干预主查询匹配文档的评分。

12.3.3 Block Join

除了上面两种 Join 方式, Solr 中还支持 Block Join 方式实现 Join 查询, 但前提是你的 Document 必须是 Nested Document (即嵌套 Document)。那到底什么是嵌套 Document 呢? 所谓嵌套 Document 即你可以在一个 Document 内部再添加一个 Document, 形成父子关系, 就好比 HTML 里 <div> 标签内部能够嵌套其他 <div> 标签, 形成多层级的父子关系。拿用户和订单为例进行说明。一般可能会创建 user 和 orders 两个 Core 对两者进行单独存储, 而假如此时采用嵌套 Document 来进行索引, 此时你的索引数据结构应该是类似这样的:

```
<add>
<doc>
  <field name="id">1</field>
  <field name="user_name">张三</field>
  <field name="age">25</field>
</doc>
...// 省略
</doc>
<doc>
  ...// 省略
</doc>
</doc>
```

此时 schema.xml 应该如下进行设计:

```
<field name="id" type="long" indexed="true" stored="true" />
<field name="user_name" type="string" indexed="true" stored="true" />
<field name="age" type="tint" indexed="true" stored="true" />
<field name="orders_id" type="long" indexed="true" stored="true" />
<!-- 消费金额 -->
<field name="amount" type="tfloat" indexed="true" stored="true" />
<!-- 交易发生时间 -->
<field name="occur_date" type="tdate" indexed="true" stored="true" />
<!-- 嵌套 Document 必须添加 _root_ 域 -->
<field name="_root_" type="string" indexed="true" stored="false"/>
```


其中 `_root_` 域是嵌套 Document 必须添加的, 当你在 `schema.xml` 中添加了 `_root_` 域, 任何 Document 都可以添加子 Document 形成嵌套 Document。此外, 父 Document 的所有子 Document 必须一起添加, 而且父子 Document 不能单独更新, 即你不能单独更新父 Document 或子 Document, 你只能更新整体。关于本章节的测试 Core "user" 相关的配置文件和测试数据请从随书源码中获取。

现在让我们来思考下, 假如我想查询用户李四最近 2 周内的交易记录, 我该如何查询呢? 此时我们可以使用 Block Child Query Parser 来实现。Block Child Query Parser 的语法如下所示:

```
q={!child of=<allParents>}<someParents>
```

`{!child...}` 语法用于返回子 Document。其中 `<allParents>` 表示任意的查询表达式, 用于返回部分父 Document, 注意这里说的父 Document 仅仅只是一个概念, 只是主观上认为 `<allParents>` 查询表达式匹配到的索引文档是父 Document, 也就是说 `<allParents>` 表达式可以指定为子 Document 的域来进行过滤。后面的 `<someParents>` 也是一个查询表达式, 用于过滤前一步 `<allParents>` 表达式返回的父 Document, 从而进一步过滤掉一些想排除掉的父 Document, 最后返回符合要求的父 Document 下包含的子 Document。

了解了基本语法之后, 我们就可以开始实现我们一开始提出的需求, 以下是查询示例:

```
http://localhost:8080/solr/user/select?q={!child of="age:[20 TO 28]" }user_name:李四 &fq=occur_date:[NOW-14DAYS TO *]
```

需要注意, `<allParents>` 表达式必须是用来过滤出所有的父 Document, 否则返回的结果集可能不是你所期望的。比如当你将 `<allParents>` 表达式修改为 `age:28`, 不采用范围限时, 此时它匹配的并不是所有父 Document, 返回的结果集却十分诡异, 请执行以下的查询示例自行感受下:

```
http://localhost:8080/solr/user/select?q={!child of="age:28"}user_name: 李四
```

你发现返回的结果集却包含了张三的交易记录, 我表示费解, 为此我还特意在 Solr6.2.1 最新版本 (我编写本章时最新版本是 6.2.1) 上进行了测试, 发现还是有这种问题, 因此也就侧面证明了, Block Join 的语法本身要求 `<allParents>` 部分必须返回所有父 Document。

再考虑下面一个查询, 假如我希望查询出 [2016-08-01T00:00:00Z TO 2016-10-01T00:00:00Z] 这段时间内消费总额超过 1000 的用户, 此时应该怎么查询呢? 因为我们需要返回的是用户, 而用户是父 Document, 因此此时需要使用 `{!parent...}` 语法来实现, `{!parent...}` 的具体语法表达式如下所示:

```
{!parent which=<allParents>}<someChildren>
```

`<allParents>` 表示一个查询表达式, 用来过滤出索引中所有父 Document, `<someChildren>`

表示通过子 Document 的域来进一步限制 <allParents> 表达式返回的父 Document。因此，我们上面的需求可以这样实现，如下所示：

```
http://localhost:8080/solr/user/select?q={!parent which="age:[20 TO 28]"}occur_
date:[2016-08-01T00:00:00Z TO 2016-10-01T00:00:00Z] AND _query_:={!frange l=1000
u=1200 v=sum(amount)}
```

以上的查询中我们首先通过 which 参数里指定的查询条件将用户的年龄限制在 [20 TO 28] 之间，过滤出所有的父 Document，然后再按照用户的交易记录时间来过滤，也就是 occur_date:[2016-08-01T00:00:00Z TO 2016-10-01T00:00:00Z] 这部分，这里是利用子 Document 里的域来限制父 Document，符合这个时间范围的内的用户只有张三和王五，我们通过 !frange 语法定义了一个 Function 范围查询，我们先通过 sum(amount) 统计用户在指定时间区间内的总消费金额，然后通过 frange 中的 l 和 u 参数定义的区间限定消费金额在 [1000, 1200] 之间，符合这个要求的只有王五。

但是，你会发现我们每次仍然只能单独返回父 Document 或者子 Document，能不能实现类似 SQL 里的 select a.xxx,b,xxx 这样的功能呢？即在返回结果集中同时返回父 Document 的域以及子 Document 的域呢？答案是肯定的，此时需要使用 ChildDocTransformerFactory，它的使用语法如下所示：

```
[child parentFilter=doc_type:book childFilter=doc_type:chapter limit=100]
```

parentFilter 用于过滤父 Document，此参数必须指定，且必须匹配所有父 Document，与之前所说的 <allParents> 类似。childFilter 参数用于过滤 parentFilter 参数匹配的所有父 Document 下的子 Document，若 childFilter 参数未指定，默认会返回 parentFilter 参数匹配的所有父 Document 下的子 Document，limit 参数用于限制每个父 Document 下最多能够返回多少个子 Document。因此假如想要返回用户张三以及他下面的所有交易记录，可以执行下面的查询示例，如下所示：

```
http://localhost:8080/solr/user/select?q={!term f=user_name v=$qq}&qq=张 三
&fl=id,user_name,[child parentFilter="age:[20 TO 28]" limit=10]
```

首先通过 parentFilter 过滤出了所有的父 Document，即返回所有父 Document 下的子 Document，但是我们在 q 参数中通过 {!term...} 语法构造了一个 TermQuery，f 表示在 user_name 域上执行查询，v 表示 Term 的值，引用的是自定义的一个变量 qq，即等价于 user_name: 张三。此时返回的结果集如下所示：

```
"response":{"numFound":1,"start":0,"docs":[
  {
    "id":"1",
    "user_name":"张三",
    "_childDocuments_":[
      {
        "id":"4",
```

```

      "amount":725.4,
      "occur_date":"2016-09-30T08:00:00Z"},
    {
      "id":"5",
      "amount":199.0,
      "occur_date":"2016-10-02T11:00:00Z"},
    {
      "id":"6",
      "amount":490.5,
      "occur_date":"2016-10-18T09:10:00Z"}]]}]

```

正如上面的结果集所示，现在是返回了父 Document 下的所有子 Document。假如此时我想要继续对张三下的子 Document 进行过滤呢，比如我只想要返回单笔消费金额大于 500 的交易记录呢？此时需要再添加一个 childFilter，查询示例如下所示：

```

http://localhost:8080/solr/user/select?q={!term f=user_name v=$qq}&qq=张 三
&fl=id,user_name,[child parentFilter="age:[20 TO 28]" childFilter="amount:[500 TO
*]" limit=10]

```

在以上的查询示例中，我们添加了 childFilter，通过 amount:[500 TO *] 将返回的子 Document 的 amount 值限定在大于等于 500 的范围内，只有符合这个条件的子 Document 才会被返回。但是需要注意的是你只能返回子 Document 的所有域，并不是单独限定返回子 Document 的个别域。

此外你还可以通过 JSON Facet API 对嵌套 Document 分别进行 Parent 和 Children Facet，示例如下所示：

```

http://localhost:8080/solr/user/select?q=*&
json.facet={
  genres : {
    type: terms,
    field: user_name,
    domain: { blockParent : "age:[20 TO 25]" }
  }
}

```

我们对父 Document 按照 user_name 域进行分组统计，然后通过 domain 中的 blockParent 表达式限定返回的每个 Facet 的 age 值必须在 [20, 25] 之间，注意：blockParent 只能根据父 Document 拥有的域进行限定，比如你不能使用 blockParent:"amount:[1000 TO *]" 对父 Document 进行限定，因为 amount 是子 Document 的域。同理还有 blockChildren，由于此功能还不太成熟，我就不再演示了。如果你想要在 Block Join 查询中同时使用 Facet 进行查询统计，此时推荐使用 Solr 中的 Block Join Facet 查询组件，不过此组件要求 Solr 版本不低于 5.5。

12.3.4 Block Join Facet

首先让我们回顾下 Solr 中是如何处理结构化数据的，Solr 支持对多层级具有父子关系

的嵌套 Document 使用 Block Join Query 组件（简称 BJQ）进行查询。这种查询组件需要使用特殊的索引方式来索引文档，BJQ 依赖于索引文档在索引中的 position（位置）信息，属于同一个层级的索引文档将会被连续索引，子 Document 在前，父 Document 在后，如下所示：

```
child1,child2,,child3,parent
```

BJQ 是多个层次索引文档之间的桥梁，它会将子 Document 匹配转换成父 Document 匹配，当使用 BJQ 进行查询时，我们提供了一个子查询以及一个 Parent 过滤器（即 parentFilter），子查询表示想要查找的 Document 在这些子 Document 中间，parentFilter 告诉 BJQ 如何区分父 Document 和子 Document。对于每个匹配的子 Document，BJQ 会扫描前面的索引文档，直到找到离得最近的那个父 Document，然后将其传递给 collector（文档收集器）。但是这一切都完全依赖索引文档在索引中的相对位置，这是高性能的 BJQ 背后的原理。

现在让我们看看如何对 BJQ 之后返回结果集进行 Facet 统计。我们将使用 "clothes" 这个 Core 进行查询测试（请从随书源码中获取 Core 的配置文件和测试数据导入类）。比如我们想要统计某件衣服的颜色有几种，尺码有几种。要实现这类需求，我们使用 Solr 中内置的 BlockJoinFacetComponent 查询组件或者 BlockJoinDocSetFacetComponent 组件，BlockJoinDocSetFacetComponent 组件性能相对较高，但是要求 Facet 域必须添加 docValues="true"。

```
// 使用 BlockJoinFacetComponent 组件
http://localhost:8080/solr/clothes/blockJoinFacet?q={!parent which="type:parent"}
type:child&wt=json&indent=true&
facet=true&child.facet.field=color&child.facet.field=size

// 使用 BlockJoinDocSetFacetComponent 组件
http://localhost:8080/solr/clothes/blockJoinDocSetFacet?q={!parent
which="type:parent"}type:child&wt=json&indent=true&
facet=true&child.facet.field=color&child.facet.field=size
```

返回的结果集如下所示：

```
"facet_counts":{"facet_queries":{},
  "facet_fields":{
    "color":["Blue",1, "Red",1],
    "size":["XL",1]}}
```

对于上面的需求，还可以使用 JSON Facet API 来实现，但是 Block Join Facet 性能更好，不过 Block Join 要求索引文档必须是嵌套 Document，当然 Block Join Facet 要求 Solr 至少 5.5 版本，这也是你需要考虑的一个因素。使用 JSON Facet API 的具体查询示例如下所示：

```
http://localhost:8080/solr/clothes/select/?q=(type:child)&rows=0&
json.facet={
  colors:{
    type : terms,
    field : color,
    facet : {
```

```

        productsCount: "unique(_root_)"
      }
    },
    sizes: {
      type: terms,
      field: size,
      facet: {
        productsCount: "unique(_root_)"
      }
    }
  }
}

```

12.4 深入 Solr 相关性评分

之前我们学习了如何打开 Solr 的调试模式来显示 Solr 是如何计算相关性评分的，但是当你发现索引文档的相关性评分计算有问题时，我们应该怎么处理呢？Solr 提供了很多种方式允许你去修改索引文档的相关性评分计算，你可以很方便地调整 Term、Field、Document 的权重。所有这些的权重调整可以分别是索引时或查询时进行。

12.4.1 Field 权重

在某些索引文档中，可能某些域的权重你认为比其他域重要。比如你有个跟 Product(产品)有关的 Document，那么 product_name(产品名称)域可能要比 description(产品描述)域的权重要高，因为用户极有可能会根据产品名称进行搜索。类似的，如果你的索引文档表示的是社交网络的，那么用户名称相比于他的朋友列表域可能权重会较高。

在索引时，你可以在将 Document 发送给 Solr Server 之前，对域权重进行设置，比如：

```

<add>
<doc>
<field name="id">1</field>
<field name="restaurant_name" boost="10.0">Red Lobster</field>
...// 省略部分
</doc>
</add>

```

由于你提升了 restaurant_name 域的权重，那么当根据 restaurant_name 域进行查询时，返回的结果集中该域所在 Document 的权重就会相应增加，从而能够更靠前地显示。

尽管可以在索引时为索引文档的每个域设置不同的权重，但是这里存在几个问题，如果你修改了个别域的权重值，那么它会带来什么影响呢？如果想要修改某个域的权重值，那么就意味着你需要重建索引，这是一件很恐怖的事情。此外，索引时的域权重值是存储在内部的 fieldNorm 中，它会将域的权重值与文档权重值以及域长度一起压缩成一个字节。因此，索引时设置域的权重要求你的域的 omitNorms 必须设置为 false，否则设置无效。而且

权重值可能并不十分精确，因为它是与其他值一起压缩成一个字节。正由于这些限制，因此在查询时设置域的权重可能会是更好的选择。

```
// 为指定域增加权重
http://localhost:8080/solr/your-core/select?defType=edismax&q=red lobster&
qf=restaurant_name^10 description
```

在查询时为域增加权重不要求你在 `schema.xml` 中将域的 `omitNorms` 设置为 `false`，而且查询时设置权重值也不会索引结构中存储域的权重值，即不会为每个域额外浪费 1 个字节的空间。

12.4.2 Term 权重

有时候，你希望为查询中的某个域的个别 Term 增加权重，它跟查询时为域设置权重的语法类似，但是此时权重值是直接应用在 Term 身上。以下是几个设置示例：

```
http://localhost:8080/solr/your-core/select?
q=restaurant_name:(red^2 lobster^8)
OR description:(red^2 lobster^8)
```

那如果同时为域和域的查询 Term 设置了权重值，此时查询 Term 的权重值是怎么计算的呢？

```
http://localhost:8983/solr/no-title-boost/select?
defType=edismax&q=red^2 lobster^8&
qf=restaurant_name^10 description
```

比如这里我们先为 "red"Term 的权重值设置为 2，"lobster"Term 的权重值设置为 8，然后设置 `restaurant_name` 域的权重值为 10，那么最终 "red"Term 的权重值为 $2 \times 10 = 20$ ，而 "lobster"Term 的权重值为 80。

12.4.3 Payload 权重

如果你想要在索引时为每个 Term 单独设置权重，那么你可以给那些 Term 单独创建一个域，并为其设置更高额权重值。但是更好的方式是使用 Solr 中提供的 Payload 功能。当 Solr 将索引文档中的每个 Term 存储到索引中时，你可以为每个 Term 附带一个字节数组来存储额外的信息，这些信息在查询时可能会比较有用。Term 携带的字节数组被称为 Payload。Payload 值可以用于索引文档打分阶段来影响文档最终的相关性评分。在索引时解析 Payload 信息你需要借助 Solr 中提供的 `DelimitedPayloadFilterFactory` 来完成。以下是一个 `DelimitedPayloadFilterFactory` 的简单配置实例：

```
<fieldType name="text_dlmtd" class="solr.TextField" positionIncrementGap="100">
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
```



```
<filter class="solr.DelimitedPayloadTokenFilterFactory" encoder="float"
delimiter="|"/>
</analyzer>
</fieldType>
```

但是当前 Solr 不支持在查询时传递 Payload 信息来影响索引文档的相关性评分。不过你可以自定义 Query Parser 或者自定义 Similarity 类来进行扩展实现。关于这方面的扩展你可以上 Github 搜索 "Payload Query Parser", 有部分开源代码供你参考, 这里就不展开说明了。

12.4.4 Function 权重

在 11.1 章节中我们介绍了 Solr 中的 Function 查询功能, 在本小节中, 我们将学习如何使用 Function 计算并将函数计算值作为相关性评分模型中的一部分。开始之前, 请首先根据随书源码提供的文件创建好 "distance-relevancy" 和 "news-relevancy" 这两个测试 Core 并导入测试数据。

Solr 中的 Function 会认为每个 Term 都能匹配所有索引文档, 并将函数计算值作为相关性评分。这意味着通过将 Function 注入到你的 Query 中, 你就可以操纵 Solr 的相关性评分算法, 甚至替换它, 从而实现索引文档的评分的重新设计以满足你的实际需求。请执行以下这个查询示例:

```
http://localhost:8080/solr/distance-relevancy/select?
q=restaurant_name:(Burger King) AND
_query_:"{!func}recip(geodist(location,37.765,-122.43),1,10,1)"
```

以上的查询示例中首先查询 restaurant_name 域中包含 "Burger King" (汉堡王) 的索引文档, 然后计算每个索引文档的评分, 粗略等于 score("Burger") + score("King") + 函数计算值。recip 函数会为离 (37.765, -122.43) 坐标点位置比较近的索引文档权重增加 10, 这样离的比较近的索引文档在查询结果集中的权重就较高。recip 函数其实就好比数学里的倒数函数, 除了可以通过距离远近来为索引文档增加权重之外, 还可以根据距离某个时间点的时间长短来为索引文档增加权重, 此时你需要使用 ms 函数来获取时间的毫秒数, 具体请翻到第 11 章重新回顾 Solr 中的内置函数。OK, 请继续考虑以下这个查询示例:

```
http://localhost:8080/solr/news-relevancy/select?
fq={!cache=false v=$keywords}&
q=_query_:"{!func}scale(query($keywords),0,100)"
AND _query_:"{!func}div(100,map(geodist(location,$pt),0,1,1))"
AND _query_:"{!func}recip(rord(publicationDate),1,100,1)"
AND _query_:"{!func}scale(popularity,0,100)"&
keywords="street festival"&
pt=33.748,-84.391
```

以上查询示例中, 我们分别通过 popularity 域 (人气值) 从高到低、publicationDate 域 (出版时间) 从远至近、location 域 (地理位置) 从远至近、\$keywords 关键字查询的相关性,

这 4 个维度去分别对索引文档进行增加权重。当你希望将人气较旺的索引文档增加权重使其靠前显示, 比如畅销书你可能希望优先显示。当希望新出版的书籍优先靠前显示时, 你可以对表示出版时间的域使用 `rord` 函数来进行计算权重。有时候你可能更希望返回离你比较近的索引文档信息时, `geodist()` 函数会对你比较有用, 比如实现搜索离用户比较近的酒店、饭店、书店等等功能需求时, 上面的示例是个很好的参考。通常上面的查询, 与查询关键字更相关、人气值更高、离我们更近、出版时间离当前时间更近的索引文档将会被返回。但是为你的查询添加 Function 能够多维度的为你的索引文档进行加权, 不过需要注意添加的 Function 过多, 可能会导致查询执行性能问题, 你需要在查询性能和查询返回的索引文档相关性之间做好平衡。

12.4.5 邻近 Term 权重

如果发现你的短语查询返回的查询结果集中只是包含了查询 Term 中的个别 Term, 那么此时可能需要为你的查询请求设置邻近 Term 权重来提升你的查询返回结果集的相关度。

当执行一个查询请求, Solr 会根据向量空间模型对每个 Term 进行相关度计算, 然后对所有 Term 的得分进行求和, 默认查询中 Term 的邻近信息并没有纳入评分计算。这意味着当执行一个查询 `content:(statue of liberty)`, 包含 "statue"、"of"、"liberty" 的索引文档都会被返回, 但是包含 "statue of liberty" 这个完整短语的索引文档的评分并没有比其他索引文档高。有时候, 你可能想要为包含邻近 Term 的索引文档增加权重, 从而使其尽量靠前显示, 比如搜索 "iPhone 6 Plus", 那么那些仅仅只是包含了 "iphone" 这个 Term 的索引返回了, 而完整包含 "iPhone 6 Plus" 这个 Term 短语的索引文档也返回了。你可能更希望完整包含你输入的 Term 短语的索引文档权重更高, 应该优先靠前显示。

在 Solr 中为邻近 Term 增加权重一般有两种主要方式: 使用 `eDisMax Query Parser` 或者使用默认的 `Lucene Query Parser`。

在 `eDisMax` 中首先需要通过 `pf` (`phrase field` 的缩写) 参数来为指定域启用邻近 Term 加权。`pf` 参数的语法为:

```
field~slop^boost
```

使用示例如下所示:

```
/select?defType=edismax&
q=big data analytics&qf=content&
pf=content~3^10
```

除了 `pf` 参数, `eDisMaxQueryParser` 还支持定义额外的 `pf` 参数, 比如 `pf2` 和 `pf3` 参数。假如上面示例中, 我们指定了 `pf2=content`, 那么就表示 Solr 会对包含 "big data" 和 "data analytics" 这样的双 Term 组合短语的索引文档进行加权。同时还有个 `ps2` (`ps` 即 `phrase slop` 的缩写, 这里的 2 表示 2-gram) 参数用于控制两个 Term 之间间隔多少个间距就认定为

2-gram。同理还有 pf3 参数，即表示对包含 "big data analytics" 这样的 3 个 Term 组合短语的索引文档进行加权，配套的还有 pf3 参数。通过这些参数的设置，你能够对那些包含邻近 Term 的索引文档进行加权，从而使得你的查询相关度更高。

当然你也可以使用传统的 Lucene Query Parser 来设置邻近 Term 的权重，在 lucene 中设置邻近 Term 权重的语法为：

```
"term1 term2 ... termN"~slop^boost
```

举个例子，比如搜索 "customer service"~2^10，表示返回索引文档中包含 "customer" 或者 "service" 这两个 Term 中任意一个的索引文档，同时假如索引文档中包含 "customer service" 或者 "customer XXX service" 或者 "customer XXX XXXX service" 这样的 Term，那么会将这些索引文档的权重值设置为 10。

以下是几个设置示例：

```
1 q=+customer +service OR "customer service"
2 q=+customer +service OR "customer service"~2^10
3 q=customer OR service +{some other terms}
+(*:* OR "customer service"^100)
4 q=+customer +service +representative
OR "customer service representative"^10
```

第一个示例中，任何同时包含 "customer" 和 "service" 的索引文档会被返回，但是如果索引文档中精确包含 "customer service"，此时索引文档的相关性会提升 1 倍。"customer service" 等价于隐式的 "customer service"^1，更具体点，就是等价于 "customer service"~0^1。

第二个示例与第一个示例类似，不过显式地指定了两个 Term 之间最大可以有 2 个间隔，2 个间隔之内的都认为是邻近 Term，然后会为包含该邻近 Term 的索引文档权重设置为 10。

第三个示例演示了如何为某个特定的 Term 短语设置权重，表示它们必须在索引文档中紧挨在一起出现，并且为那些包含该 Term 短语的索引文档加权。

第四个示例中需要注意一点，你从中可以了解到，你可以为任意个 Term 组成的短语进行权重设置。

虽然你可以通过以上介绍的两个方式来为邻近 Term 增加权重，同时你需要清楚，邻近 Term 权重的设置需要额外的时间去解析，虽然它能提升你的查询相关度，但是它同时也有可能影响你的查询响应速度。

12.4.6 Document 权重

为索引文档增加权重有 3 种主要方式，其中最简单的方式就是在索引时为索引文档设置权重，这种方式与在索引时为域设置权重类似，示例如下所示：

```
<add>
<doc boost="10.0">
```

```
<field name="id">1</field>
...// 其他省略
</doc>
</add>
```

如果你是使用 SolrJ 的方式添加 Document, 那么此时你可以通过 SolrInputDocument 对象提供的 setDocumentBoost() 方法来为索引文档设置权重。

在 Solr 内部, 索引时的索引文档级别权重和域级别的权重很相似。这意味着索引时为索引文档设置权重等价于为索引文档中的每个域都设置一个相同的权重值。因此, 它与索引时的域级别拥有同样的问题, 一般也不建议在索引时设置索引文档权重。至于为什么, 我们已经在 Field 权重部分讨论过了。对于域级别的权重我们建议是在查询时为域设置权重, 但是你可能会有这样一个疑问: 是不是也可以在查询时为索引文档设置权重? 答案是肯定的。

另外一种比较灵活的方式就是为索引文档添加一个 popularity (表示人气) 域, 然后在查询时针对指定域执行一个 Function, 根据 Function 计算值来为每个索引文档进行加权。不管是索引时设置权重还是查询时通过 Function 来设置权重, 目的都是一个: 为索引文档设置权重, 只是设置时机不同, 添加 popularity 域只不过是將索引文档的权重值设置延迟到查询时。

Solr 还支持一种查询组件, 它能允许人为的为特定的查询设置某些索引文档放到查询结果集的顶部显示或者屏蔽部分索引文档不显示, 它就是 Query Elevation 组件。通过 Query Elevation 组件你能任意调整索引文档的权重, 而直接无视相关性评分规则。关于 Solr 中的 Query Elevation 查询组件更详细的内容请翻到第 11 章的第 12 小节进行回顾学习。

12.4.7 自定义 Similarity 插件

在 Solr 中 (准确来说, 是在 Lucene 中), 默认的相关性评分采用的是 TF-IDF 向量空间模型余弦相似度计算方式实现。最近几年, Solr 中陆陆续续引进了多种可选的 Similarity 实现。在 Solr 中不仅可以为每个 Schema 设置一个 Similarity 实现, 还可以单独为某个域设置 Similarity 实现, 配置示例如下所示:

```
<similarity class="solr.DFRSimilarityFactory">
  <str name="basicModel">I(F)</str>
  <str name="afterEffect">B</str>
  <str name="normalization">H2</str>
</similarity>
```

如果自定义的 Similarity 实现没有任何输入参数, 那么你可以这样设置:

```
<similarity class="my.package.MyCustomSimilarity" />
```

单独为某个特定域设置 Similarity 实现, 配置示例如下所示:

```
<fieldType name="text_custom_similarity" class="solr.TextField">
```

```

<analyzer>
    ...// 省略
</analyzer>
<similarity class="my.package.MyCustomSimilarity" />
</fieldType>

```

如果默认的 DefaultSimilarity 评分实现不能满足你的需求，那么此时可以考虑 Solr 中提供的其他评分实现，表 12-1 详细列举了 Solr 中自带的各种 Similarity 评分实现。

表 12-1 Solr 中的各种 Similarity 评分实现

函 数	描 述
DefaultSimilarity	Solr 中索引文档的默认评分实现，基于 TF-IDF 外加更多标准化因子实现
BM25Similarity	基于概率模型的 TF-IDF 相关性评分实现，基于 BM25 算法，Solr6 中已经将默认评分实现修改为 BM25Similarity
DFRSimilarity	DFR 评分公式根据 Term 在每个索引文档中相对其他索引文档的出现频率的相对分部来决定一个 Term 在索引文档中的相对重要性
IBSimilarity	IB (Information-Based 的缩写) 评分公式一种类似于 DFRSimilarity 的新相似度算法，但是输入更简单
LMDirichletSimilarity	使用语言模型，它提供了一个贝叶斯滤波 Term 权重语料库
LMJelinekMercerSimilarity	同样是使用了语言模型，但是评分实现比 LMDirichletSimilarity 更简单
SweetSpotSimilarityFactory	一个 DefaultSimilarity 的扩展，它提供了调优参数来决定最佳的 TF 和 lengthNorm

如果以上的所有内置 Similarity 评分实现都不能满足你的需求，此时你可能需要自定义 Similarity 评分实现了。

在 Solr 中自定义 Similarity 实现，大致需要以下几个步骤：

- 1) 继承 Solr 中的 DefaultSimilarity 类，并重写其相关方法。
- 2) 继承 Solr 中的 DefaultSimilarityFactory 工厂类，并重写其 getSimilarity() 方法，在该方法中创建并返回第 1 步中实现的 Similarity 对象。
- 3) 将 1)、2) 步中自定义的实现类打包成 jar，然后复制到当前 Core 的 lib 目录下，并在 solrconfig.xml 中通过 <lib/> 元素导入 jar 包。
- 4) 在 schema.xml 中对自定义的 Similarity 实现类进行注册或者直接应用到某个特定域上，上面已经给出了配置示例。

以下是在 Solr 中自定义 Similarity 插件的一个简单示例，完整示例代码请从随书源码获取。

```

public class PayloadSimilarity extends DefaultSimilarity {
    private static final float[] boosts = {0.0f, 1.0f, 10.0f, 20.0f, 40.0f, 80.0f};
    private static final int maxBootsIndex = boosts.length - 1;
    public PayloadSimilarity() {}
    @Override
    public float scorePayload(int doc, int start, int end, BytesRef payload) {

```



```

        int length = Utils.bytes2Int(payload.bytes);
        // 根据掺入的 Payload 数据设置不同的权重值
        return boosts[Math.min(length, maxBootsIndex)];
    }

    @Override
    public String toString() {
        return "PayloadSimilarity";
    }
}

```

12.5 Solr NRT 近实时查询

所谓 Near Real Time (简称 NRT, 翻译过来就是近实时的意思) 查询, 就是当索引文档被索引后能几乎立即被查询到。当索引提交正在处理进行过程中, 索引的更新删除操作并不会被阻塞, 也不会等待后台的索引合并工作完成之后才为索引打开一个新的 `IndexSearcher` 实例并返回。使用 NRT 查询, 你可以将一个索引提交命令更改为软提交 (Soft Commit), 软提交能够避免标准提交的巨大代价。有时候可能确实想要使用标准提交来确保索引数据正确地写入存储介质中, 但是软提交能够让你在索引提交期间就能够近乎实时地看到新添加的索引数据。但是你需要特殊注意缓存和 Autowarm (自动预热) 的配置, 它们会严重影响 NRT 查询的性能。

一个索引提交操作使得索引更新能够对新的查询请求可见, 一个硬提交使用事务日志来获取最新的 Document ID, 同时对索引文件调用 `fsync` 方法确保索引数据写入到硬盘, 并且保证即便在断电的极端情况下也不会造成数据丢失。

索引软提交之所以比较快速, 是因为它只是使得更新的索引数据能够对查询请求可见, 但是并没有刷新硬盘上的索引文件或者在硬盘上创建任何索引文件。如果 JVM 崩溃了或者服务器断电了, 那么上一次硬提交之后的所有索引更新数据将会被丢失。如果你的查询要求索引更新在之后能够快速地被查询到, 此时你需要开启索引软提交。软提交可以频繁执行但是硬提交不行。软提交在耗时方面开销较小, 但是并不是毫无代价的, 因为它会降低你的索引吞吐量。

硬提交 (Hard Commit) 它会要求所有的段文件必须立即合并为一个段文件, 并重写整个索引, 这个操作执行开销很大, 不能执行太频繁。段文件合并应该合理配置合并策略, 并定期执行索引优化, 提升查询性能。

为了达到灵活提交目的, Solr 为硬提交和软提交设计了自动提交策略, 自动提交主要靠 `maxDocs` 和 `maxTime` 两个参数控制, `maxTime` 参数表示每间隔多少毫秒就触发一次索引提交, `maxDocs` 表示当队列中累积了多少个索引文档就触发一起索引提交。你可以在 `solrconfig.xml` 中配置自动提交策略, 配置示例如下所示:

```

<autoCommit>
  <maxTime>15000</maxTime>

```



```
<maxDocs>1000</maxDocs>
<openSearcher>false</openSearcher>
</autoCommit>
```

其中 `openSearcher` 表示当执行一次硬提交之后是否立即打开一个新的 `IndexSearcher` 实例。同理还有 `<autoSoftCommit>` 配置, `maxTime` 和 `maxDocs` 参数同样适用于 `<autoSoftCommit>` 软提交。但是对于软提交配置 `maxTime` 更合理, 尤其是当索引大量的索引文档时。当批量索引时, 如果仅仅只是为了追求索引吞吐量, 并不要求索引数据能立即被查询到, 那么你可以关闭软提交。通常来说, 对于硬提交一般建议是每间隔 1-10 分钟执行一次, 对于软提交一般建议是每间隔 1 秒钟执行一次。通过这样配置, 保证了新添加的索引文档能够秒级被查询到, 同时假如突然断电了, 除非硬提交来不及写完新更新的索引数据, 否则软提交并不会丢失数据。

在 Solr 中除了可以通过开启软提交来实现近实时查询, 还可以通过传递 `<commit/>` 指令的方式实现近实时查询。我们知道, 在索引文档时, XML 文件中可以通过 `<add>`、`<delete>` 元素来指示 Solr 是添加还是删除索引文档, 还可以通过添加 `<commit>` 元素来指示 Solr 执行一次硬提交, 同理还有 `<optimize/>` 用于指示 Solr 执行一次索引优化。关于这部分内容, 在 2.2.2 章节有做介绍。索引优化其实可以看作一种特殊的索引 Commit 操作, 它除了需要做索引 Commit 之外, 还会执行索引合并, 所以它的执行开销比硬提交还大。

当你在待索引的 XML 文件中添加 `<commit>` 元素, 或者直接 Http 请求发送 `<commit/>` 指令, 或者直接通过请求参数显式指示 Solr 是硬提交还是软提交时, 默认是执行硬提交。`<commit/>` 和 `<optimize/>` 元素都支持以下 3 个参数:

- ❑ `waitFlush` 表示会阻塞提交请求, 直到索引写入硬盘之后才会返回。默认为 `true`。不过此参与已经不建议使用了。
- ❑ `waitSearcher` 表示直到一个新的 `IndexSearcher` 实例创建并注册完成之后才返回, 否则一直阻塞提交请求。
- ❑ `softCommit` 表示是否使用软提交方式提交索引, 通过此设置, 可以开启近实时搜索, 但是不能保证索引真正写入硬盘。

`<commit/>` 元素还有一个独有的 `expungeDeletes` 参数, 表示清除被标记为删除的索引文档并合并段文件, 默认值 `false`。`<optimize/>` 元素也有一个独有的 `maxSegments` 参数, 表示段文件合并后的段文件最大个数, 默认值 "1"。以下是配置示例, 如下所示:

```
<commit softCommit="true" waitSearcher="false" expungeDeletes="true"/>
<optimize waitSearcher="false"/>
```

你可以通过在请求 URL 中添加 `softCommit` 参数并设置为 `true` 来开启索引软提交模式, 从而开启 Solr 的近实时搜索, 示例如下所示:

```
http://localhost:8080/solr/update?optimize=false&openSearcher=true&waitSearcher=true&expungeDeletes=false&softCommit=true
```

但是这种方式要求你在 `solrconfig.xml` 中配置 `<autoCommit>`。你也可以发送如下请求,

指示 Solr 执行一次索引优化，并将段文件合并为 10 个文件，并阻塞等到合并操作完毕。

```
http://localhost:8080/solr/update?optimize=true&maxSegments=10&waitFlush=false
```

你甚至可以传递一个 `prepareCommit` 参数，指示 Solr 执行一次 `prepareCommit` 操作，示例如下所示：

```
http://localhost:8080/solr/update?prepareCommit=true
```

那么什么是 `prepareCommit` 呢？或者说 `prepareCommit` 和 `commit` 操作之间有什么区别？在 Lucene 中，索引提交分两个阶段提交，`prepareCommit` 操作属于第一阶段。第一阶段会做一些准备工作，比如将队列中待添加和标记为待删除的索引文档刷新到索引文件中，然后同步索引文件、创建并写入段文件，当第一阶段执行完之后，需要显式调用 `commit` 去结束提交操作，或者调用 `rollback` 去回滚提交操作。如果直接调用 `commit`，那么内部会隐式的先调用 `prepareCommit`。

除了可以显式的指定 `softCommit=true` 参数来开启对近实时查询的支持，同时还可以在索引时通过设置 `commitWithin=1500` 参数来启用 Solr 的近实时查询。`commitWithin` 参数需要设置一个毫秒数，表示告诉 Solr 请在规定毫秒数之内提交索引文档，然后 Solr 会将用户提交的索引文档放入队列，当达到规定的最大时间后，Solr 会触发一次索引硬提交。也就是说当客户端用户执行 `commitWithin` 操作时默认是软提交，直至达到限制的最大时间，Solr 会自动触发一次硬提交，从而保证在限制的时间内将索引文档写入。通过 `commitWithin` 方式并不需要你在 `solrconfig.xml` 中配置 `<autoCommit>`，因为达到用户设置的最大时间后，Solr 会自动触发一次硬提交。

12.6 Solr Real-time Get 查询

除了上一章节中提到的 3 种方式实现 NRT（近实时）查询之外，Solr 还提供了 Real-time Get API 来实现实时 GET 请求。Real-time Get API 允许实时获取最新版本的一个或多个索引文档。这里使用的是 Real-time，意思就是真正的实时查询，为的就是区分传统的 NRT 查询。传统的基于 Lucene 的近实时查询一直都是通过周期性的刷新一个版本的索引快照的方式来保证近实时查询。而 Real-time Get 会保证实时的返回最新版本的索引文档，而且 Real-time Get 实现实时查询不需要重新打开一个新的 `IndexSearcher` 实例。但是，Real-time Get 有个局限性：只能根据 Unique-Key（唯一性主键）进行查询。

当想要将 Solr 当作一个 NoSQL 数据存储系统，而不仅仅只是索引查询，此时 Real-time GET 会很有用。但 Real-time GET 依赖于 Solr 的 Update Log（事务日志）功能，要想使用 Real-time GET，必须先启用 Solr 的 Update Log 功能。Solr 的 Update Log 默认就是启用的，你可以在 `solrconfig.xml` 中对其进行配置，示例如下所示：

```
<updateLog>
```

```
<str name="dir">${solr.ulong.dir:}</str>
</updateLog>
```

然后你还需要在 solrconfig.xml 中配置 RealTimeGetHandler 请求处理器, 配置示例如下所示:

```
<requestHandler name="/get" class="solr.RealTimeGetHandler">
<lst name="defaults">
<str name="omitHeader">true</str>
<str name="wt">json</str>
<str name="indent">true</str>
</lst>
</requestHandler>
```

下面具体演示下 Solr 中的 Real-time GET 查询, 首先我们创建一个 "real-time-get" Core(请从随书源码中获取相关配置文件), 然后配置 schema.xml, 添加 id 和 name 两个域, 就简单地使用 string 域类型即可, 为了保证演示效果, 我们先将 solrconfig.xml 中的 <autoCommit> 和 <autoSoftCommit> 配置都注释掉, 防止发生自动提交。Real time Get 测试代码如下, 通过执行如下代码你能切身感受到 Real time Get 的强大:

```
private static final String SOLRPEDIA_INSTANT_CORE = "http://localhost:8080/
solr/real-time-get";
HttpSolrClient client = new HttpSolrClient(SOLRPEDIA_INSTANT_CORE);
client.setRequestWriter(new BinaryRequestWriter());
UpdateRequest request = new UpdateRequest();
// 这个用于设置硬提交
// request.setAction(UpdateRequest.ACTION.COMMIT, true, true);
// 设置在 2 万秒之后, 自动触发一次软提交, commitWith 默认是软提交,
// 软提交方式默认的查询方式是获取不到最新的索引数据的
request.setCommitWithin(20000000);
SolrInputDocument doc = new SolrInputDocument();
doc.addField("id", "1");
doc.addField("name", "This is a Real-time Get Test1.");
request.add(doc);
System.out.println(request.getXML());
NamedList<Object> result = client.request(request);
System.out.println("Result: " + result);
SolrQuery query = new SolrQuery();
// Real time GET 查询测试
query.setRequestHandler("/get");
query.set("id", "1");
query.set("fl", "id,name");
QueryResponse response = client.query(query);
System.out.println("以下是 Real time GET 的响应信息: \n");
System.out.println(response.toString());
// 普通 select 查询测试
query.setRequestHandler("/select");
query.set("q", "id:1");
query.set("fl", "id,name");
```

```
response = client.query(query);
System.out.println("\n 以下是普通 select 查询请求的响应信息: \n");
System.out.println(response.toString());
```

RealTimeGetHandler 实时查询组件支持如下几个请求参数:

- id: 你想要获取的最新的索引文档的 id 值。
- ids: 逗号分隔的索引文档 id 值, 用于批量实时获取多个最新的索引文档。
- fl: 逗号分隔的域名称列表, 即你需要返回索引文档的哪些域。
- fq: 即添加一个 Filter Query, 对 Real time Get 请求返回的索引文档进行再次过滤。



注意 如果在 SolrCloud 分布式环境下, 请你不要禁用 RealTimeGetHandler, 否则 Leader 选举时会导致分片上所有索引副本执行一次全量索引同步, 同样的, 副本恢复也会总是从 Leader 节点上拉取整个索引数据, 因为当 RealTimeGetHandler 未配置时, 部分同步不可用。

12.7 Solr 评分查询

自 Solr4.9 版本开始, Solr 提供了 Ranking Query (评分查询) 来干预另一个普通 Query 返回的索引文档的最终评分。你执行一个普通查询 (这里简称为 A), 然后会返回 Top N 个索引文档, 然后可以再为 A 指定一个更复杂的 Ranking Query (这里简称为 B), B 会返回一个分数, 然后 A 的每个 TOP N 索引文档的最终评分 = A 的每个 TOP N 索引文档的原始评分 + 权重因子 * B 返回的评分。因此 B 的执行开销会比较大, 所以一般将其应用于 TOP N 结果集上, 这样对查询性能影响较小。如果某个索引文档的原始评分很低, 即便当使用了 B 之后, 该索引文档的评分会变得很高, 但是此时不会对该索引文档进行重新评分。

索引文档的 Re-Ranking (重新评分) 其实是采用两阶段评分机制, 复杂的评分计算通常是在第二阶段进行, 只有限定的 Top N 索引文档会参与第二阶段的评分计算。

我们可以使用 rq (即 Ranking Query 的缩写) 参数来指定 Ranking Query, rq 参数应该包含一个查询表达式, 用于构造一个 RankQuery。Solr 中提供了 ReRank Query Parser 来帮助你更加简单的构造一个 RankQuery。它可以通过指定一个本地参数来包装一个 Query, 你还可以指定额外的参数来指示至少多少个索引文档应该被重新评分以及为 RankQuery 返回的评分增加权重。下面表格详细列举了 Solr 中的 ReRank Query Parser 支持的设置参数, 如表 12-2 所示。

表 12-2 ReRank Query Parser 参数表

函 数	描 述
reRankQuery	此参数必须指定, 表示复杂的 Ranking Query 查询表达式, 通过做法是通过一个变量引用另外一个请求参数

(续)

函 数	描 述
reRankDocs	表示原始查询中的 Top N 索引文档中多少个索引文档应该重新评分, 这个数值表示的最小值, 内部实际会根据指定的 start 和 rows 参数来自动增长此参数值, 以便能够重新评分足够的索引文档来满足查询。参数默认值为 200
reRankWeight	这是一个乘法因子, 它会应用到 Ranking Query 返回的评分上。默认值 2.0

以下是一个 Ranking Query 简单使用示例, 如下所示:

```
?q=Digital group&rq={!rerank reRankQuery=$rqq reRankDocs=1000reRankWeight=3}&r
qq=solrcloud
```

——以上的查询首先查询所有包含 "Digital group" 的所有索引文档, 然后对 Top 1000 索引文档进行重新评分, 同时对还包含了 "solrcloud" 的索引文档的加权因子设置为 3。

Ranking Query 是一种动态的为索引文档进行评分的比较好的方式, 尽管 Ranking Query 可能会有一些性能问题, 但是在有些时候, 它还是很有用的。

这里我们借用 12.3.4 章节里的 "user"Core 对 Ranking Query 进行演示, 以下是一个 Ranking Query 简单使用示例, 如下所示:

```
http://localhost:8080/solr/user/select?q=age:[20 TO 28]
&rq={!rerank reRankQuery=$rqq reRankDocs=1000 reRankWeight=10}&rqq=age:[* TO 20]
```

——以上这个查询示例中, 我们先使用 age:[20 TO 28] 过滤出符合要求的用户, 然后根据后面的 age:[* TO 20] 查询 (即年龄小于 20 的用户) 对我们的主查询进行加权, 意思就是如果主查询符合要求的前 1000 个索引文档中哪个索引文档满足 age:[* TO 20] 条件的, 就将其索引文档增加权重, 权重增加控制因素由 reRankWeight 参数以及 Ranking Query 返回的分数一起决定。我们的示例中, 符合 age:[* TO 20] 条件的只有用户 "王五", 因此最终返回的结果集中用户 "王五" 排在第一位。当然你可以构造更复杂的查询, 来对主查询返回的索引文档进行重新评分。

12.8 Solr MoreLikeThis 组件

当你查询返回与你输入关键字相关的索引文档, 你可能还希望返回与当前返回的索引文档相似的索引文档, 比如你搜索 "Java", 系统返回了 Java 相关的技术书籍, 同时你可能还希望返回跟 Java 类似的书籍, 比如 "JavaScript" 相关书籍。此时你可能需要使用 Solr 中的 MoreLikeThis 查询组件。它使用 Term 在原始的索引文档中查询相似的索引文档。

在 Solr 中使用 MoreLikeThis 查询组件有 3 种方式, 首先最常用的方式就是将其注册为 Request Handler (请求处理器)。第二种方式就是将其注册为 Search Component (查询组件), 这种方式不建议使用, 因为它会为返回的每个索引文档执行 MoreLikeThis 分析操作, 会影响

查询性能。最后一种方式也是将其注册为 Request Handler，但是，使用的是外部提供的文本。

MoreLikeThis 会基于索引文档中的 Term 构造一个 Lucene Query，而这些 Term 需要从定义的域列表上获取，该域列表需要通过 mlt.fl 参数执行，为了获取最佳效果，那些域应该存储 Term 向量信息，即域的 termVectors 属性需要设置为 true，配置示例如下所示：

```
<field name="cat" ... termVectors="true" />
```

如果域的 Term 向量信息未存储，那么 MoreLikeThis 会自动从存储域（stored=true 的域）上生成 Term。为了使 MoreLikeThis 正常工作，你还必须存储 UniqueKey。

表 12-3 列举了 MoreLikeThis 的 3 种使用方式都支持的通用参数。

表 12-3 MoreLikeThis 参数表

参 数	描 述
mlt.fl	用于计算相似度的域名称列表，多个域名称使用逗号分隔。请尽量将这些域的 termVectors 属性设置为 true，但不是强制性的
mlt.mintf	指定 Term 的最小频率，Term 在某个索引文档中的出现频率小于此参数值，那么该 Term 将会被忽略
mlt.mindf	指定索引文档的最小频率，Term 所在索引文档的个数小于此参数值，那么该 Term 将会被忽略
mlt.maxdf	指定索引文档的最大频率，Term 所在索引文档的个数大于此参数值，那么该 Term 将会被忽略
mlt.minwl	设置词的最小长度，小于此参数值的词将会被忽略
mlt.maxwl	设置词的最大长度，大于此参数值的词将会被忽略
mlt.maxqt	用于指定在任意生成的 Query 中查询 Term 的最大个数，超过的将被忽略
mlt.maxntp	表示从没有设置 termVectors=true 的存储域上获取 Token 的最大个数
mlt.boost	表示应该使用有趣 Term 的相关性来对查询进行加权
mlt.qf	用于指定查询域，这些域同样需要在 mlt.fl 参数中指定

MoreLikeThisComponent 组件支持如下额外的参数，如表 12-4 所示。

表 12-4 MoreLikeThisComponent 参数表

参 数	描 述
mlt	表示是否激活 MoreLikeThis 查询组件，启用后，Solr 会返回 MoreLikeThis 的查询结果集
mlt.count	指定每个索引文档返回的相似索引文档最大个数，默认值为 5

MoreLikeThisHandler 请求处理器支持如下额外的参数，如表 12-5 所示。

表 12-5 MoreLikeThisHandler 参数表

参 数	描 述
mlt.match.include	用于指定是否返回 MoreLikeThis 匹配的索引文档，如果设置为 false，那么看起来就跟普通的 select 一样
mlt.match.offset	用于指定主查询结果集的偏移量来定位被 MoreLikeThis 查询所操作的索引文档。默认 MoreLikeThis 查询会操作主查询的第一个索引文档
mlt.interestingTerms	用于控制 MoreLikeThis 应该如何显示有趣的 Term，可选值有 3 种： list 表示列出所有有趣的 Term；none 表示不显示任何 Term；details 表示显示 Term 以及 Term 的权重值。如果 mlt.boost=false，那么所有有趣 Term 的权重值都是默认值 1.0

为了简化用户构造 MoreLikeThis 查询，Solr 提供了 MLTQParser 语法解析器用于构造 MoreLikeThis Query。MLTQParser 允许你为给定的 Document 返回一批相似的 Document。它使用 Lucene 内置的 MoreLikeThis 逻辑，并且 MoreLikeThis 查询支持 SlorCloud 模式。MoreLikeThis 需要使用唯一主键 ID 域来标识 Document，而不是 Lucene Document 内部的 ID。表 12-6 列举了 MLTQParser 语法解析器支持的配置参数。

表 12-6 MLTQParser 语法解析器支持的配置参数

参 数	描 述
qf	用于指定查询域，同 MoreLikeThis 的 mlt.qf 参数
mintf	同 MoreLikeThis 的 mlt.mintf 参数
mindf	同 MoreLikeThis 的 mlt.mindf 参数
maxdf	同 MoreLikeThis 的 mlt.maxdf 参数
minwl	同 MoreLikeThis 的 mlt.minwl 参数
maxwl	同 MoreLikeThis 的 mlt.maxwl 参数
maxqt	同 MoreLikeThis 的 mlt.maxqt 参数
maxntp	同 MoreLikeThis 的 mlt.maxntp 参数
boost	同 MoreLikeThis 的 mlt.boost 参数

在 Solr 中使用 MoreLikeThis 第一种方式就是在 solrconfig.xml 中注册 MoreLikeThisHandler 请求处理器，配置示例如下所示：

```
<requestHandler name="/mlt" class="solr.MoreLikeThisHandler">
</requestHandler>
```

然后按照随书源码创建好 "morelikethis"Core 并导入测试数据，然后你可以执行如下查询进行 MoreLikeThis 查询测试：

```
http://localhost:8080/solr/morelikethis/mlt?
df=book_name&fl=id,book_name&rows=20&
wt=json&indent=true&q=hadoop&
mlt.fl=book_name&mlt.mintf=1&mlt.mindf=1&
mlt.match.include=true&
mlt.interestingTerms=book_name
```

返回的结果集如下所示：

```
"match":{"numFound":3,"start":0,"docs":[
  {
    "book_name":"Mastering Hadoop",
    "id":"18"}}
},
"response":{"numFound":2,"start":0,"docs":[
  {
    "book_name":"Hadoop in action",
    "id":"16"},
```

```
{
  "book_name": "Hadoop in Practice",
  "id": "17"}}
```

其中 match 部分返回的是最佳匹配的 Document，并根据这个 Document 生成 interesting term。docs 下返回的是 MoreLikeThis 返回的与 Match 部分返回的 Document 相似的 Document。

在 Solr 中使用 MoreLikeThis 的第二种方式就是在 solrconfig.xml 中注册 MoreLikeThisComponent 组件，但是由于 MoreLikeThisComponent 查询组件默认已经注册到 SearchHandler 中，因此你不需要做任何配置。你唯一需要做的就是发起查询请求时传递 mlt=true 参数来开启 MoreLikeThisComponent 查询组件，以下是关于 MoreLikeThisComponent 查询组件的使用示例：

```
http://localhost:8080/solr/morelikethis/select?
q=book_name:hadoop&mlt=true&mlt.fl=book_name&mlt.count=3&
mlt.mintf=1&mlt.mindf=1
```

MoreLikeThisComponent 查询组件会为每个索引文档都返回指定数量的相似索引文档，因此它的计算量相对要大一些。

Solr 中使用 MoreLikeThis 的第 3 种方式，其实和第 1 种方式是类似的，只不过此时查找相似文档的参照体变了，第一种方式是通过 q 参数构造的主查询然后返回一个 Document，然后与这个 Document 相比较计算相似度，从而为其返回相似文档。而第 3 种方式是通过与你提供的一段外部数据源来进行比较计算相似度，该外部数据源可以是你指定的一段字符串文本，可以是 HTTP URL 表示的网页里的文本内容。第 3 种方式和第 1 种方式配置一样。下面是几个使用示例：

通过 stream.body 参数指定一段字符串文本，然后在 Solr 索引中查找与该指定字符串文本相似的索引文档：

```
http://localhost:8080/solr/morelikethis/mlt?
stream.body=Is there a story for the Hadoop Storage Stack (HDFS+HBase) on
Solid State Drive (SSD)?&
fl=id,book_name&rows=3&
wt=json&indent=true&
mlt.fl=book_name&mlt.mintf=1&mlt.mindf=1&
mlt.match.include=true&
mlt.interestingTerms=details
```

通过 stream.url 参数指定一个 HTTP URL，然后在 Solr 索引中查找与该指定 URL 表示的网页文本内容相似的索引文档：

```
http://localhost:8080/solr/morelikethis/mlt?
stream.url=https://databricks.com/blog/2016/08/31/apache-spark-scale-a-60-tb-
production-use-case.html&fl=id,book_name&rows=3&wt=json&indent=true&
mlt.fl=book_name&mlt.mintf=1&mlt.mindf=1&
mlt.match.include=true&mlt.interestingTerms=details
```

由于这种方式需要通过网络连接去获取网页内容，可能会由于网络状况而导致查询响应时间过长，所以一般不建议使用。

12.9 Solr 自定义 Query Parser

你有没有遇到这种情况，当执行一个 Solr 查询时，你可能期望关键字被前缀匹配了，那么就应该为该索引文档增加权重，使其优先靠前显示。举例说明，假如你有类似以下的索引文档：

```
"iPhone 6S Plus",
"Apple iPhone 6S Plus",
"Apple iPhone 6S",
"iPhone 6S",
"iPhone 6",
"iPhone 6S White 64G",
"Apple iPhone 6S Plus Black 64G",
"Apple iPhone 6 Plus White 64G"
```

当搜索 "iphone 6"，你可能希望 "iPhone 6" 这个索引文档排在第一位，因为它精确匹配了你的搜索关键字，然后第二位应该返回 "iPhone 6S"，因为它的前缀与你的搜索关键字相匹配。此时你可能考虑通过自定义 Query Parser 来实现。下面我们来自定义一个 PrefixBoostQParser，比如输入 "^iphone" 就表示需要前缀匹配，然后自动加权，你也可以手动设置权重值，比如 "^iphone"^20。同理，输入 "iphone^" 就表示后缀匹配，输入 "^iphone^" 即表示精确完整匹配。

在 Solr 中，自定义 Query Parser 需要继承 org.apache.solr.search.QParser 类，然后重写其 parse() 方法。你可以直接从父类继承到 qstr、localParams、solrParams、request 等属性信息，其中 qstr 表示查询文本。比如我们自定义的查询表达式语法是 {!prefixBoost qf=name exactname^10}"^iphone 6S"^10，那么 "^iphone 6S"^10 部分就表示 qstr，即 {!.....} 外部的文本，也有可能 {!.....} 外部没有定义文本，此时 qstr 等于 {!.....} 内部定义的参数 v 的值。localParams 表示 Solr 中的本地参数，关于 Solr 中的 LocalParams 本地参数相关知识点请翻到 5.3 章节进行回顾。其中 solrParams 属性表示 Solr 查询请求参数，即你在请求 URL 中传递的参数，可以是 Solr 内置定义的参数，也可以是你自定义的任何参数，比如 &qq1=xxxxx&&qq2=xxxxx，在 Query Parser 类的内部可以通过 solrParams 获取到这些参数值，request 属性表示 Solr 查询请求对象，通过它你可以获取 schema 的相关信息，比如域名称、域名称、域所使用的分词器、查询表达式默认 Boolean 连接符、默认查询域等等。通过上面这几个属性获取到的信息来构建 Query 对象。关于自定义的 PrefixBoostQParser 类完整源码请从随书源码中获取。

Query Parser 自定义完成之后，还需要自定义 QParserPlugin，用于在 solrconfig.xml 中

注册。自定义 QParserPlugin 首先需要继承 org.apache.solr.search.QParserPlugin 类，然后重写其 createParser 和 init 方法，createParser 方法中创建我们刚刚自定义的 PrefixBoostQParser 类对象即可，这样自定义的 PrefixBoostQParserPlugin 插件就完成了。你需要在 solrconfig.xml 中注册，配置示例如下所示：

```
<queryParser name="prefixBoost" class=
"com.yida.solr.book.examples.ch12.queryparser.PrefixBoostQParserPlugin"/>
<requestDispatcher handleSelect="false" >
...// 其他省略
```

然后你需要仿照下面的配置示例在 schema.xml 中定义搜索域：

```
<fieldType name="text_lowercase" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory" />
  </analyzer>
</fieldType>
<fieldType name="text_lower_exact" class="solr.TextField"
  positionIncrementGap="100" sortMissingLast="true" omitNorms="true">
  <analyzer>
  <charFilter class="solr.PatternReplaceCharFilterFactory"
    pattern="^(.*)$" replacement="æ $1 Æ" />
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory" />
  <filter class="solr.PatternReplaceFilterFactory"
    pattern="([^\a-z0-9æøåäöéúæÆ ])" replacement="" replace="all" />
  </analyzer>
</fieldType>
<field name="name" type="text_lowercase" indexed="true" stored="true"
  omitNorms="true" />
<field name="exactname" type="text_lower_exact" indexed="true" stored="false"
  omitNorms="true" />
<field name="_version_" type="long" indexed="true" stored="true" />
<copyField source="name" dest="exactname"/>
```

其中 exactname 域的域值从 name 域复制，exactname 域用于前缀匹配或者精确匹配。为了使得前缀匹配时权重值更高，可以指定 qf 参数来分别为搜索域设置权重值，比如 qf="name exactname^10"。然后你就可以执行如下查询进行测试：

```
http://localhost:8080/solr/prefixboost/select?q={!prefixBoost qf="name exactname^10"}^iphone 6S"^10 "^iphone 6S"^100 "^iphone 6"^20&
wt=json&indent=true&fl=id,name,score&debug=true
```

其中 {!prefixBoost...} 这里的 prefixBoost 就是我们在 solrconfig.xml 中注册的 name 属性值，Solr 会根据这个名称从而知道构造哪个 Query Parser 来对查询表达式进行解析。

12.10 本章总结

在本章中，首先我们学习了如何通过深度分页来提供 Solr 分页查询的性能，但是同时你要谨记 Solr 深度分页的局限性。然后我们学习了在 Solr 中如何实现自定义排序来满足特定复杂的需求。接着我们学习了 Solr 中的 Join 查询以实现跨 Core 跨文档的连接查询，但是它并不完全等同于 SQL 中的 join 查询，它们各有各的使用场景和限制。紧接着我们通过各种方式来设置权重从而提升查询的相关性。然后我们学习了 Solr 中的近实时查询和实时 GET 查询，以及 Solr 中的 MoreLikeThis 组件，该组件可返回相似文档推荐给用户，关于用户个性化推荐方面建议使用更专业的 Apache Mahout 框架来实现，最后我们学习了如何在 Solr 中自定义 Query Parser。

通过第 13 章，你将可以学习到以下内容：

- 什么是 SolrJ；
- SolrJ 的环境依赖与配置；
- SolrClient 介绍；
- 如何使用 SolrJ 添加更新删除索引；
- 如何使用 SolrJ 查询 Solr 索引数据；
- 如何使用 SolrJ 高效导出 Solr 索引数据；
- 如何使用 SolrJ 增量更新索引；
- 如何使用 SolrJ 原子更新索引；
- 如何使用 SolrJ 管理 Core 和 Schema；
- 如何使用 SolrJ JSON Request API；
- 如何使用 Sparing-Data-Solr。

13.1 什么是 SolrJ

SolrJ 是 Java 客户端访问 Solr Server 的 API，它提供了一套完整的 Java API 接口用于往 Solr 中添加、更新、删除索引以及从 Solr 中查询数据。SolrJ 帮用户屏蔽了大量客户端连接 Solr Server 的详细信息，它允许你的应用程序使用简单易用的 API 接口与 Solr 进行数据交互。

我们知道 Solr Server 一般是部署在 Web 容器中，然后通过 HTTP 协议进行网络通信，因此可以通过浏览器来请求 Solr Server，当然也可以通过 Java 里的 URLConnection 类来

连接 Solr Server, 甚至可以借助 Apache HttpClient 来模拟 HTTP 请求 (其实 SolrJ 就是基于 HttpClient 封装的), 然后 Solr Server 返回 JSON/XML 格式的数据, 我们还需要自己去做数据解析, 然而, 这一切 SolrJ 已经帮你封装在 "黑盒子" 里, 你不用关心内部是如何实现与 Solr Server 交互通信的, 只需要引入它并使用它即可简单实现与 Solr Server 交互。而且 SolrJ 由官方提供且经过无数用户的测试检验, 完全值得信赖, 因此我们完全没必要自己重造轮子, 这也就是为什么我们需要使用 SolrJ。

由于 Solr Server 是基于 HTTP 协议来提供 Solr 服务, 而 HTTP 协议与编程语言无关, 也就意味着你可以在大部分目前流行的编程语言中通过模拟 HTTP 请求来实现与 Solr Server 进行数据交互。不过目前市面上已经提供了几种主流编程语言相关的 Solr 客户端类库, 比如 RSolr(Ruby)、SolrNet(.NET)、Solrs(Scala)、Solrpy(Python) 等等。Java 语言相关的 Solr 客户端类库就是 SolrJ, 当然还有 Spring 提供的 Spring-Data-Solr, 不过 Spring-Data-Solr 本质其实就是基于 SolrJ 进行了一层薄封装, 并严重依赖 Spring 的 IOC 容器, 虽然它在 SolrJ 基础之上, 做了很多不错的封装和扩展, 但是不免也带有一点硬性推广自身产品的嫌疑, 然而也不可否认, Spring 产品在国内公司已经到处在渗透, 不使用 Spring 的 Java 项目几乎很少, 所以如果你的项目中有添加 Spring 依赖, 可以考虑下使用 Spring-Data-Solr。

13.2 SolrJ 的环境依赖与配置

想要在 Java 程序中使用 SolrJ, 首先你需要导入 SolrJ 依赖的 jar 包, SolrJ 依赖的 jar 包在 Solr 的 /dist、/dist/solrj-lib、/lib 等目录下。使用 SolrJ 的最小依赖如下所示:

/dist 目录下:

```
apache-solr-solrj-*.jar
```

/dist/solrj-lib 目录下:

```
commons-codec-version.jar
commons-httpclient-version.jar
commons-io-version.jar
jcl-over-slf4j-version.jar
slf4j-api-version.jar
```

/lib 目录下:

```
slf4j-jdk14-version.jar
```

请根据上述提示将 Jar 包复制到你项目的 Classpath 路径下。当然, 如果你的项目是根据 Maven 构建的, 那么你可以直接通过在 pom.xml 中添加 SolrJ 依赖即可, 配置示例如下所示:

```
<properties>
```

```
<solr.version>5.3.1</solr.version>
</properties>
<dependency>
<groupId>org.apache.solr</groupId>
<artifactId>solr-solrj</artifactId>
<version>${solr.version}</version>
</dependency>
```

如果你的 Solr 并不是部署在 Web 容器中，而是直接嵌入到当前项目中，那么你将与 Solr Server 之间的通信就不是基于 HTTP 协议了，而是直接通过 Solr Java API 去交互即同一个项目内简单的 Java 方法调用而已。此时需要使用 SolrContainer 类来加载 SOLR_HOME 目录并加载所有 Solr Core，Solr Core 加载完毕之后你就可以通过 SolrContainer 对象和 Core 名称来构建 EmbeddedSolrServer 对象，通过 EmbeddedSolrServer 对象你就可以与 Solr 进行数据交互。而由于 SolrContainer 和 SolrCore 这些类在 solr-core-version.jar 包内部，因此，在这种使用场景下，你还需要额外添加如下 jar 包依赖：

```
<dependency>
<artifactId>solr-core</artifactId>
<groupId>org.apache.solr</groupId>
<version>${solr.version}</version>
<type>jar</type>
<scope>compile</scope>
</dependency>
```

我们知道大部分的 Web 容器都是基于 Http Servlet API 来支持 HTTP 请求，这要求你添加 servlet-api-2.5.jar 依赖，不过一般 Web 容器下都包含了这个 jar 依赖，对于 Java Web Project 来说，不需要添加此依赖。但是如果当前你的项目仅仅只是简单的 Java Project，那么此时你还需要显式的添加 Servlet-api 依赖，配置示例如下所示：

```
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>servlet-api</artifactId>
<version>2.5</version>
</dependency>
```

如果你想要在开发时能够显示有关 Solr 的详细日志信息，那么你还需要添加 slf4j 依赖，具体 slf4j 依赖的版本号请查看 dist\solrj-lib 目录下依赖的 slf4j-api jar 的版本号，配置示例如下所示：

```
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-api</artifactId>
<version>${slf4j.version}</version>
</dependency>
```

当然你还需要将 log4j.properties 配置文件放到当前项目的 Classpath 路径下，否则无法

打印日志。如果使用过程中还提示 `NoClassDefFoundError` 依赖, 那么请按照异常提示确定缺少的类存在于哪个 Jar 包, 然后添加该依赖即可。

SolrJ 通常具有很强的兼容性, 因此可以使用新版本的 SolrJ 来与旧版本的 Solr Server 进行数据交互, 或者使用旧版本的 SolrJ 与新版本的 Solr Server 进行数据交互。但是使用 SolrJ 时需要注意, 能使用 SolrJ4.x 与 Solr3.x Server 进行数据交互, 因为默认的 Request Writer: `javabin` 不兼容 Solr3.x。一般建议 SolrJ 与 Solr Server 主版本保持一致即可。

13.3 SolrClient 介绍

SolrJ 相关的类都包含在 *Solr* 源码的 `org.apache.solr.client.solrj` 包下, 该包下面主要包含了 5 个关键类: `SolrClient`、`SolrRequest`、`SolrQuery`、`SolrResponse`、`ResponseParser`。其中 `SolrClient` 类表示 Solr Client 是一个 Solr 客户端的抽象。通过 `SolrClient` 你可以与 Solr Server 进行如下操作:

add: 添加一个索引文档到 Solr Server, 为了支持面向对象编程, `SolrClient` 提供了 `addBean` 方法, 允许直接提交一个 Java 对象, `SolrClient` 内部会借助 `DocumentObjectBinder` 来完成 Java 对象与 `SolrInputDocument` 对象之间的类型转换。为了支持批量添加索引文档, 你还可以直接 `add` 一个 `Document` 集合或者通过 `addBeans` 方法添加一个对象集合。

commit: 即显式的提交一个索引硬提交请求。

optimize: 即显式的要求 Solr Server 执行一次索引优化, 会触发 Solr 的索引段文件合并, 此操作一般不建议调用太频繁。

rollback: 即显式的要求 Solr Server 将尚未提交的索引文档从队列中剔除, 此回滚操作并不像是关系型数据库里的回滚操作, 它不保证 100% 会回滚, 因为你上一次提交的索引文档可能会由于你配置了 `AutoCommit`, 可能刚好由于缓冲区满了导致一次索引 Flush、或者 `AutoCommit` 的间隔时间或者最大缓存索引文档个数达到临界值自动触发了一次硬提交, 或者另一个 `SolrClient` 客户端显式的发起了 `commit` 操作导致索引已经提交, 此时索引文档已经写入到硬盘, `rollback` 操作将无济于事。

deleteById & deleteByQuery: 这两个方法主要用于向 Solr Server 发起一个索引删除操作, `deleteById` 表示根据 Document 的主键 ID 来删除索引文档, 这里的主键 ID 是你在 `schema.xml` 中配置的 `uniqueKey`。 `deleteByQuery` 用于按照 Solr 查询来删除多个索引文档。

query & getById: 这组方法主要用于向 Solr Server 发起一个查询请求, 其中 `getById` 方法是一种特殊的查询请求, 即根据索引文档的主键 ID 进行查询, 前提是你的 `schema.xml` 配置了 `uniqueKey`。

ping: 用于发起一个 ping 请求来检测 Solr Server 是否还“活着”, 前提是你需要在 `solrconfig.xml` 中配置 `PingRequestHandler`。

request: 用于向 Solr Server 的指定 Core 或 Collection 发起一个 HTTP 请求。

shutdown & close：释放 SolrClient 使用过程中被分配的所有资源，shutdown 方法已经标记为废弃，不推荐使用。

使用 SolrJ 之前，首先需要构建一个 SolrClient 实例，然后发送一个 Solr 请求，SolrRequest 是 Solr 请求的抽象，如果需要执行 Solr 查询，那么你需要构建一个 SolrQuery 实例，SolrQuery 是 Solr 查询的抽象，最终 Solr Server 会返回一个响应信息给客户端，SolrResponse 是这里的 Solr 响应信息的抽象。其中 SolrResponse 响应信息需要使用 ResponseParser 来解析转换成 NamedList 类型返回给客户端。SolrClient 实际被设计为一个抽象类，在使用时，首先需要创建一个 HttpSolrClient 或者 ConcurrentUpdateSolrClient 或者 LBHttpSolrClient 或者 CloudSolrClient 实例。各种 SolrClient 实现子类之前的区别，请翻到 10.8.2 章节进行回顾，这里不再赘述。

在 HttpSolrClient 内部维护了一个 HttpClient 实例，每次 HttpSolrClient 构建时都会判断内部的 HttpClient 实例是否为空，如果为空才会新建 HttpClient 实例对象。这样做的目的是保证每个 HttpSolrClient 实例只对应一个 HttpClient 实例，由于这里的 HttpClient 实例由 final 关键字修饰，因此它是线程安全的。但是如果你重新 new 一个 HttpSolrClient 实例，那么可能又会重新创建一个 HttpClient 实例，然而创建 HttpClient 实例需要一定的执行开销，比如重新初始化 ClientConnectionManager（客户端连接管理器）。因此一般建议 HttpClient 实例采用单例模式，也就意味着 SolrClient 实例也最好是采用单例模式，尽量在请求结束之后，在 finally 代码块内显式调用 close 方法释放资源。但是对于多个 HTTP 请求，请在多个 HTTP 请求完成之后再关闭 HttpClient，而不是请求一次，关闭然后再打开。而 ConcurrentUpdateSolrClient 类内部是直接维护一个 HttpSolrClient 实例，在其构造函数内部直接 new HttpSolrClient 实例，因此，ConcurrentUpdateSolrClient 对象也应该保持单例，况且 ConcurrentUpdateSolrClient 内部维护了一个请求队列和线程池，如果每次重新创建 ConcurrentUpdateSolrClient 实例意味着需要重新分配队列空间和构建线程池，每次都单个 Http 请求构建一个队列和线程池，那么这是一种空间浪费。LBHttpSolrClient 和 CloudSolrClient 同理。综上所述，强烈建议每个 Solr Server URL 对应一个静态的 SolrClient 实例来处理所有请求，所有请求处理完毕再关闭。如果你每次都临时动态创建 SolrClient 实例，可能会造成 Http 连接泄漏。

HttpSolrClient 实例有几个可以优化的关键点，如下所示：

setConnectionTimeout：设置 Http 连接的最大超时时间（单位：毫秒）。

setSoTimeout：设置 Http Socket 读操作最大超时时间。

setFollowRedirects：设置是否开启 Http 自动重定向，所谓 "Http 自动重定向" 表示当 Http Server 返回一个 302 响应状态码并将网页内容指向一个新的 URL 时，是否自动跳转到新的 URL。一般为了防止出现网页劫持，默认会禁用 Follow Redirect。

setAllowCompression：设置是否允许响应数据压缩，目前支持 gzip 和 deflate 两种压缩方式。但前提是你的 Http Server 支持响应数据压缩。此方法只对 DefatulHttpClient 类型设置有效。

`setDefaultMaxConnectionsPerHost`：用于设置单个主机在任意时刻可以打开的最大 Http 连接数。但前提是 `HttpClient` 对象是由内部自动构建的而不是外部传入的。

`setMaxTotalConnections`：用于设置在任意时刻可以打开的最大 Http 连接数，但前提是 `HttpClient` 对象是由内部自动构建的而不是外部传入的。

`isUseMultiPartPost`：`MultiPart Post` 一般用于文件上传，这里用于判断是否为文件上传请求，比如你可能会直接上传一个 XML 文件到 Solr Server 来创建索引。

`setUseMultiPartPost`：设置当前是一个文件上传请求。

13.4 SolrJ 简单使用

创建 `SolrClient` 实例：

```
private static final String SOLRPEDIA_INSTANT = "http://localhost:8080/solr/";
// 可以先不设置请求哪个 Core
SolrClient client = new HttpSolrClient(SOLRPEDIA_INSTANT);
// 可以在查询时临时指定请求哪个 Core
QueryResponse response = client.query("core1", solrQuery);
// 或者通过 request 方法临时设置请求哪个 Core, query 和 request 方法二选一即可
NamedList<Object> result = client.request(new QueryRequest(solrParams), "core1");
// 你还可以直接在构造 HttpSolrClient 实例时就明确指定请求哪个 Core
client = new HttpSolrClient(SOLRPEDIA_INSTANT_CORE + "core1");
```

创建索引文档：

```
SolrInputDocument doc = new SolrInputDocument();
doc.addField(fieldName, fieldValue);
doc.setField(fieldName, fieldValue);
```

`addField` 和 `setField` 的区别：`addField` 内部会判断该域是不是多值域，如果是多值域，对同一个域多次调用 `addField` 会追加域值，而 `setField` 不管你是什么域，一律覆盖域值。当域不是多值域时，`addField` 和 `setField` 没有什么区别。当你明确知道当前设置的域不是多值域时，可以使用 `setField`，但是如果当前设置的域是多值域或者你压根不清楚当前域是不是多值域，那么此时应该使用 `addField`。

添加索引文档：

```
client.add(doc); // 添加单个索引文档
client.add(docList); // 批量添加多个索引文档
```

你还可以直接添加一个 `JavaBean`，比如你有这样一个普通 Java 类：

```
public class Weibo {
    @Field
    private Integer id;
    @Field
    private String title;
```

其中 `@Field` 注解用于 Java 类属性与 Solr `schema.xml` 中定义的域名称之间的映射，如果两者名称不一致，你可以使用 `@Field("title_s")` 的方式来明确指定域名称。然后你就可以通过 `addBean` 方法添加一个 Java 对象，以面向对象的方式来添加索引文档（但是，切记要记得为该类提供无参构造函数）：

```
client.addBean(new Weibo()); // 单个添加
client.addBeans(weiboList); // 批量添加多个
```

提交索引文档：

```
client.commit(); // 显式发起一个索引硬提交请求
```

`commit` 方法还有两个输入参数：`waitFlush` 和 `waitSearcher`，`waitFlush` 参数表示当前提交操作一直阻塞，直到索引文档写入硬盘。`waitSearcher` 参数表示当前提交操作一直阻塞，直到一个新的 `IndexSearcher` 实例被打开并注册，并且索引更新已经可见。两个参数不设置默认都是 `true`。

索引硬提交还有另外一种方式：

```
UpdateRequest request = new UpdateRequest();
// 这里的两个 boolean 参数也是用于设置 waitFlush 和 waitSearcher
// ACTION.COMMIT 表示执行硬提交操作
request.setAction(UpdateRequest.ACTION.COMMIT, true, true);
NamedList<Object> result = client.request(request);
```

索引文档软提交：

```
UpdateRequest request = new UpdateRequest();
// 设置在 2 万秒之后，自动触发一次软提交，commitWith 默认是软提交，
// 软提交方式默认的查询方式是获取不到最新的索引数据的
request.setCommitWithin(20000000);
request.add(document);
NamedList<Object> result = client.request(request);
```

采用软提交方式提交索引文档之后，如果反悔了，还可以通过 `rollback()` 方法回滚，但是不保证 100% 回滚成功。但是采用硬提交方式是无法回滚的。可以定期的调用 `optimize` 方法执行索引优化操作，但是调用频率不要太高，因为索引优化是一个执行开销很大的操作。

索引文档其实并不支持更新操作，我们知道，Lucene 中的索引更新其实就是先删除后添加，在 Solr 中要更新一个索引文档，应该首先在 `schema.xml` 中配置 `<uniqueKey>`，然后再 `add` 一个相同 `uniqueKey` 的索引文档即可，Solr 会自动根据 `uniqueKey` 域的值来判断是“更新”还是新增。如果想要实现单个域的更新呢？此时你需要使用 Solr 中的原子更新功能，但是注意，这里说的原子更新仅仅只是 API 级别的更新，Solr 里根本不存在索引更新的概念，即便只是修改了一个域，你仍然需要删除整个 Document，然后重新添加。关于 Solr 原子更新，在 2.5.6 章节有做介绍，至于如何在 SolrJ 中执行原子更新操作，请继续关注本章后续章节。

索引删除:

```
// 根据索引文档的 uniqueKey 删除索引文档, 默认为硬提交
deleteById(String id)
// 你还可以通过指定多个 uniqueKey 批量删除多个索引文档
deleteById(List<String> ids)
// 你还可以显式指定删除哪个 collection 上的索引文档,
// commitWithinMs 参数用于延迟多少毫秒后触发一次软提交
deleteById(String collection, String id, int commitWithinMs)
// 根据指定的查询表达式删除索引文档
deleteByQuery(String query)
```

不管是添加索引文档、更新索引文档、还是删除索引文档, 其实底层本质都是生成 `<add>`、`<update>`、`<delete>` 等 XML 指令的方式来实现, 比如 `deleteByQuery("name:solr")` 最终会生成如下的 XML 指令:

```
<delete>
<query>name:solr</query>
</delete>
```

关于 `<add>`、`<update>`、`<delete>`、`<commit>` 等 XML 指令在 2.2.2 章节也有所说明。如果了解这些 XML 指令生成规则, 你完成可以自己生成, 然后通过 `stream.body` 参数将生成的 XML 指令字符串传递给 Solr Server 即可实现与 Solr 的各种数据交互。

还可以通过上传 XML/JSON/CSV 文件方式来创建索引文档, 但是前提是你需要在 `solrconfig.xml` 中的 `<requestParsers>` 元素中添加 `enableRemoteStreaming="true"`。细心的你, 应该已经发现我们已经多次在添加测试 Core 数据时采用上传 XML 方式完成, 比如第 9 章随书源码的 `IndexGeospatial` 测试类中, 我们通过 `stream.file` 参数指定一个本地的 XML 文件路径, 然后 SolrJ 读取里面的 XML 指令, 该 XML 文件内的内容必须是 Solr 规定的 XML 指令格式, 具体请打开测试文件便知。当然, 还支持 JSON 格式的指令, 具体定义规则请翻到 2.2.2 章节重新回顾, 这里不再重复。你甚至可以通过 `stream.url` 参数设置远程网络上的一个 XML 文件并上传它实现索引的添加 / 更新 / 删除等操作, 只要该 XML 文件的内容是符合 Solr 规定的指令格式即可, 示例代码如下所示:

```
// 设置网络上的文件 URL, 直接上传到 Solr Server 然后添加 / 更新 / 删除索引文档
request.setParam("stream.url", "http://xxxxxxx/xxx/xxxx.xml");
request.setParam("stream.contentType", "application/xml");
NamedList<Object> result = client.request(request);
```

还可以通过 JDBC 方式从外部数据库读取数据, 然后创建 `SolrInputDocument` 对象, 最后批量提交来创建索引, 我们已经在第 12 章的自定义排序章节的导入数据测试类 `IndexWeibo` 中演示了这种用法, 请读者自由阅读随书源码进行理解。由此扩展, 你可以使用 Java IO 读取任意文件来创建索引, 通过 MongoDB Driver 连接 MongoDB 数据库读取其中数据来创建索引等不管外部数据源是以何种形式存在, 最终目的都是读取它并将其转换成

SolrInputDocument 对象。

13.5 SolrJ 查询

Solr 中的请求操作统一由 SolrRequest 进行抽象，SolrRequest 有很多子类实现，如图 13-1 所示。其中每个 SolrRequest 对应 Solr 中的一种 http 请求，每种请求都会对应一个 Request Handler 进行接收并处理。我们知道，每个 Request Handler 都需要提前在 solrconfig.xml 中通过 <requestHandler> 元素进行注册，每个 requestHandler 通过一个 name 属性值——映射，比如 /select 对应 SearchHandler，/get 对应 RealTimeGetHandler 等，而我们的每个 SolrRequest 实现都对应着一个 name 属性值，比如 QueryRequest 实现则对应着 /select，即表明将会发送一个 `http://localhost:8080/solr/core1/select?xxx` 请求，显然 UpdateRequest 就是对应着 /update，ContentStreamUpdateRequest 虽然也对应着 /update，但是它是用于上传文件的，其他 SolrRequest 实现同理。



图 13-1 SolrRequest 实现

演示如何使用 QueryRequest 实现 Solr 查询的示例代码如下所示：

```
SolrQuery query = new SolrQuery();
query.setRequestHandler("/select");
query.set("q", "type:book");
query.set("fl", "id,brand,color,size,score");
query.set("indent", "true");
// 演示 QueryRequest 如何使用
QueryRequest request = new QueryRequest(query, SolrRequest.METHOD.GET);
NamedList<Object> result = client.request(request);
```

其实，你也可以直接调用 SolrClient 的 query 方法，传入 SolrQuery 对象即可，不需要创建 QueryRequest 对象：

```
QueryResponse response = client.query(query, SolrRequest.METHOD.GET);
```

Solr 中不管是任何查询，最终体现到 URL 上面就是传递的 URL 参数不同罢了，因此，在 SolrJ 中不管是执行 Facet 查询还是高亮查询等，只需要通过 SolrQuery 对象设置相应查询需要的参数即可，比如下面这个 Facet 查询：

```
http://localhost:8080/solr/restaurants/select?q=*&rows=0&facet=true&facet.
field=name
```

显然，我们需要传递的参数就是 `q`、`rows`、`facet`、`facet.field`。Solr 中的请求参数是使用 `SolrParams` 类进行抽象的，`SolrParams` 类下方有很多子类实现，如图 13-2 所示。

一般常用的就是 `ModifiableSolrParams` 及其子类 `SolrQuery`，直接调用 `set(key,value)` 方法设置参数即可。有时候一个参数可能会有多个参数值，比如 `facet.field` 参数，可以指定多个 `facet.field` 参数即表示根据多个 field 进行 Facet 查询，此时可以使用 `set(key,value1,value2,...)` 这种方式进行设置，或者直接 `set(key,valueArray)` 设置一个数组。

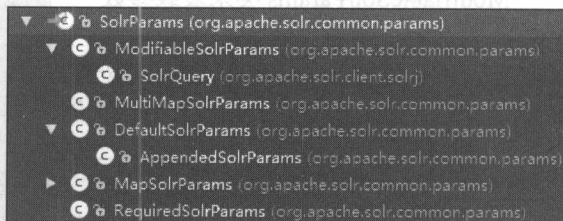


图 13-2 SolrParams 的子类实现

演示 `ModifiableSolrParams` 类如何使用的示例代码如下所示：

```
ModifiableSolrParams params = new ModifiableSolrParams();
// params.set 用于设置单个参数值，当一个参数有多个参数值时，你需要使用 params.add 方法
params.set("q", "type:book");
params.set("fl", "id,brand,color,size,score");
params.set("indent", "true");
SolrQuery query = new SolrQuery();
query.setRequestHandler("/select");
query.add(params);
```

请求参数还可以直接使用普通 `Map` 对象装载，然后传入 `ModifiableSolrParams` 类的构造函数即可：

```
Map<String,String[]> map = new LinkedHashMap<String,String[]>();
map.put("wt", new String[] {"json"});
map.put("indent", new String[] {"true"});
map.put("rows", new String[] {"10"});
map.put("q", new String[] {"*:*"});
map.put("facet.field", new String[] {"name"});
map.put("fl", new String[] {"id,name,type,city,price,score"});
// 如何使用 Map 装载请求参数
ModifiableSolrParams params = new ModifiableSolrParams(map);
```

还可以使用一个 `ModifiableSolrParams` 对象装载另一个 `ModifiableSolrParams` 对象，示例代码如下所示：

```
ModifiableSolrParams mp = new ModifiableSolrParams();
mp.add("wt", "json");
mp.add("indent", "true");
mp.add("rows", "10");
mp.add("q", "*:*");
mp.add("facet.field", "name");
mp.add("fl", "id,name,type,city,price,score");
```

```
ModifiableSolrParams params = new ModifiableSolrParams(mp);
```

ModifiableSolrParams 类来装载参数有个缺点就是你需要正确记住 Solr 中各种查询组件所支持的请求参数。当然这些请求参数你可以随时查阅得到，但是为了方便用户设置请求参数，SolrQuery 类中对一些常见的请求参数设置封装了相应的比较适用的工具方法，下面是 SolrQuery 类中提供的一些请求参数设置方法，主要包括 Facet 和 Highlighting 查询相关的查询设置，还包括 addSort 设置排序规则的，addFilterQuery 设置 fq 参数的等，如图 13-3 所示。根据提供的工具方法来设置参数使得你不需要记住每个参数名称，只需要了解每个方法用于设置什么参数即可。当然 SolrQuery 提供的工具方法并没有覆盖所有类型的查询，因此，有时候，还是避免不了需要通过 set 或者 add 或者 setParam 方法来设置参数。

演示如何使用 SolrQuery 自身的工具方法的示例代码，如下所示：

```
SolrQuery query = new SolrQuery();
query.setRequestHandler("/select");
query.addFacetField("name");
query.setFacet(true);
query.setQuery("*:*");
query.setFields(new String[] { "id", "name", "type", "city", "price", "score" });
query.setRows(10);
query.setIncludeScore(true);
```

如果调用的是 SolrClient 的 query 方法，那么最终返回的将是 QueryResponse 对象，QueryResponse 在 Solr 中表示查询最终的响应结果，通过它你能获取响应数据。表 13-1 列举了 QueryResponse 类中提供的获取各类查询组件返回的响应数据工具方法，如表 13-1 所示。

表 13-1 QueryResponse 类中提供的获取各类查询组件返回的响应数据工具方法

方 法	返回值类型	JSON 属性	描 述
getHeader	NamedList<Object>	responseHeader	返回响应头信息

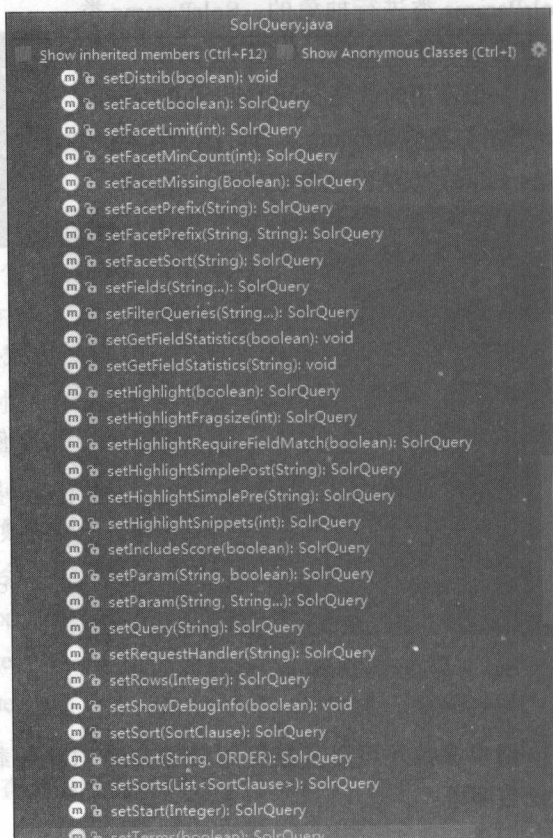


图 13-3 SolrQuery 提供的用于设置请求参数的工具方法

(续)

方 法	返回值类型	JSON 属性	描 述
getResults	SolrDocumentList	response	返回响应结果集
getSortValues	NamedList<ArrayList>	sort_values	返回排序信息
getNextCursorMark	String	nextCursorMark	返回下一个游标值, 用于深度分页
getGroupResponse	GroupResponse	grouped	返回分组查询的响应信息
getExpandedResults	Map<String, SolrDocumentList>	expanded	返回每个分组下展开的 Document
getFacetQuery	Map<String, Integer>	facet_queries	返回 Query Facet 的响应信息
getFacetFields	List<FacetField>	facet_fields	返回 Field Query 的响应信息
getLimitingFacets	List<FacetField>	facet_fields	跟 getFacetFields 类似, 但是返回的 Facet Count 不精确
getFacetDates	List<FacetField>	facet_dates	返回 facet_dates 部分响应信息
getFacetRanges	List<FacetField>	facet_ranges	返回 Range Facet 的响应信息
getFacetPivot	NamedList<List<PivotField>>	facet_pivot	返回 Pivot Facet 的响应信息
getIntervalFacets	List<IntervalFacet>	facet_intervals	返回 Interval Facet 的响应信息
getHighlighting	Map<String, Map<String, List<String>>>	highlighting	返回高亮查询相关响应信息
getSpellCheckResponse	SpellCheckResponse	spellcheck	返回 SpellCheck 组件的响应信息
getClusteringResponse	ClusteringResponse	clusters	返回 Clustering 组件的响应信息
getSuggesterResponse	SuggesterResponse	suggest	返回 Suggester 组件的响应信息
getTermsResponse	TermsResponse	terms	返回 Terms 组件的响应信息
getFieldStatsInfo	Map<String, FieldStatsInfo>	stats_fields	返回 Field 统计的响应信息
getDebugMap	Map<String, Object>	debug	返回调试相关响应信息
getExplainMap	Map<String, String>	explain	返回查询分析计划相关响应信息

下面是 Field Facet 查询的响应信息解析示例代码:

```

QueryResponse response = client.query(query);
FacetField facetField = response.getFacetField("name");
String name = facetField.getName();
int totalCount = facetField.getValueCount();
System.out.println(name + ": " + totalCount);
List<FacetField.Count> counts = facetField.getValues();
for(FacetField.Count count : counts) {
    System.out.println(count.getName() + "-->" + count.getCount());
}

```

到此, 你应该已经完全掌握了如何使用 SolrJ 来执行查询并解析获取 Solr Server 返回的响应数据。虽然执行 Solr 查询你也可以调用 SolrClient 的 request 方法, 但是 request 方法返回值是 NamedList 类型, 这意味着你需要自己解析响应数据, 然而 QueryResponse 类已经全部帮我们封装好了, 我们不需要再重复劳动, 因此一般查询时建议使用 SolrClient 的 query 方法。

13.6 使用 SolrJ 高效导出数据

Solr 提供了一种流式排序技术来支持对数以百万的索引数据进行高效排序并导出。这个功能对于以下场景非常有用：用户会话分析、分布式合并与连接、基于时间轴的数据聚合汇总、在大基数域上聚合、分布式域折叠收缩、统计排序。

想要使用这项功能，首先要求所有需要排序且导出的域必须 `docValues="true"`。然后需要你显式的在 `solrconfig.xml` 中配置一个 `SearchHandler` 并且该 `Request Handler` 只能注册 `query` 查询组件，配置示例如下所示：

```
<requestHandler name="/export" class="solr.SearchHandler">
  <lst name="invariants">
    <str name="rq">{!xport}</str>
    <str name="wt">xsort</str>
    <str name="distrib">>false</str>
  </lst>
  <arr name="components">
    <str>query</str>
  </arr>
</requestHandler>
```

注意，这个 `Request Handler` 的属性必须定义为常量，即意味着 `rq`、`wt`、`distrib` 参数不能在查询时通过 URL 参数进行覆盖，只能在 `solrconfig.xml` 中提前定义，且定义后无法修改。

一旦 `/export` 请求处理器定义好之后，你就可以开始使用它来导出索引数据。但是，导出数据的查询必须包含 `sort` 和 `fl` 参数，否则会抛出异常，导出数据查询同样支持 `Filter Query`，返回的结果集始终固定为 JSON 格式。下面是一个导出数据查询示例：

```
http://localhost:8080/solr/mapreduce/export?q=*&sort=count desc&fl=word,count
```

`sort` 参数支持对 `int`、`long`、`float`、`double`、`string`、`date`、`boolean` 等域类型进行排序，但是不支持多值域进行排序。每个数据导出请求最多只支持同时对 4 个域进行排序。`fl` 参数定义需要导出哪些域，任何能够被排序的域就可以被返回，该域可以是多值域，`socre` 这个伪域以及通配符 `*` 星号在这里不支持。使用 SolrJ 进行数据导出的示例代码，如下所示：

```
HttpSolrClient client = new HttpSolrClient(SOLRPEDIA_INSTANT_CORE);
SolrQuery query = new SolrQuery();
query.setRequestHandler("/export");
query.set("q", ":*");
query.set("fl", "word,count");
query.set("sort", "count desc");
// 设置响应数据的解析器
client.setParser(new InputStreamResponseParser("json"));
QueryResponse response = client.query(query, SolrRequest.METHOD.GET);
NamedList<Object> result = response.getResponse();
System.out.println("\n 以下是数据导出查询请求的响应信息：\n");
for(Map.Entry<String,Object> entry : result) {
```



```
String content = FileUtils.inputStream2String((InputStream)entry.getValue());
System.out.println(content + ":" + "");
}
```

13.7 SolrJ 增量更新

Solr 中内置的增量更新操作其实完全都是在 Solr Server 端完成, SolrJ 客户端只不过是
通过 Http 请求发送了一个指令罢了。实现 Solr 中的增量更新关键是需要你提前在 solrconfig.
xml 中配置 DataImportHandler, 配置示例如下所示:

```
<requestHandler name="/dataimport" class="solr.DataImportHandler">
<lst name="defaults">
<str name="config">data-config.xml</str>
</lst>
</requestHandler>
```

然后你需要配置 data-config.xml, 关于 data-config.xml 的详细配置, 请翻到第 2 章的
3-4 章节重新回顾。剩下你需要做的就是使用 SolrJ 向 /dataimport 这个请求处理器发送一个
增量更新请求, 具体的实现代码, 如下所示:

```
/**
 * 全量 / 增量导入数据
 * @param core      目标 Core 名称
 * @param entity     目标实体
 * @param delta      是否增量导入
 */
public void importData(String core,String entity,boolean delta) {
    SolrQuery query = new SolrQuery();
    // 指定 RequestHandler, 默认使用 /select
    query.setRequestHandler("/dataimport");
    String command = delta ? "delta-import" : "full-import";
    String clean = delta ? "false" : "true";
    String optimize = delta ? "false" : "true";
    query.setParam("command", command)
        .setParam("clean", clean)
        .setParam("commit", "true")
        .setParam("entity", entity)
        .setParam("optimize", optimize);

    try {
        solrClient.query(core,query, SolrRequest.METHOD.GET);
    } catch (Exception e) {
        log.error(command + "occurs an exception[core:" + core + "]:", e);
    }
}
```

不管是增量更新还是全量更新, 你都需要定时去调用上面封装的工具方法, 两者调用
时机和频率不同, 由于两者是互斥的, 因此应该尽量避免同时进行全量和增量操作。关于定

时调用，可以使用 Java 里自带的 Timer 类实现，或者借助外部的 Quartz 调度框架实现，虽然 Quartz 框架功能齐全，但是过于庞大、学习成本很高，考虑选择其他开源的定时调度实现。

13.8 SolrJ 原子更新

原子更新是自 Solr4.0 起新增的功能，允许单独更新某个域而不是更新整个索引文档。这意味着可以单独更新个别域，而不需要将整个 Document 都发送到 Solr Server 端。注意，这里说的原子更新仅仅只是 API 级别，底层其实还是先删除索引文档然后重新添加。

想要使用 Solr 中的原子更新，你必须提前在 solrconfig.xml 中配置 `<updateLog/>`。这样做是为了确保更新指令是作用于当前最新版本的索引文档上，即便该版本的索引文档仍未提交。为了提升性能，Solr 的索引文档原子更新是在 DistributedUpdateRequestProcessor 的执行期间被处理。默认情况下，在更新请求链条中 DistributedUpdateRequestProcessorFactory 的执行优先级高于 RunUpdateProcessorFactory，这对大多数用户来说没有什么影响，但是有 2 种情况下可能会影响到你：

- ❑ 如果你使用 UpdateRequestProcessor 更新索引文档，必须确保 UpdateRequestProcessor 在 DistributedUpdateRequestProcessorFactory 之后执行，这样原子更新才能接收到 DistributedUpdateRequestProcessorFactory 收集到的索引文档的所有域值。
- ❑ 如果你禁用了 DistributedUpdateRequestProcessorFactory，将其换成你自定义的 DistributingUpdateProcessorFactory 或者 NoOpDistributingUpdateProcessorFactory，那么原子更新功能也将不支持。

Solr 支持多种修改器来实现索引文档的原子更新：

- ❑ set：设置或覆盖域值，如果将域值设置为 Null 你甚至可以删除域值。
 - ❑ add：为多值域添加一个域值，只能用于多值域。
 - ❑ remove：删除一个或多个域值，用于单值域就是删除该域，用于多值域就是删除多值域的其中一个或多个值。
 - ❑ removeregex：同 remove 类似，只不过是通过正则表达式来匹配多个域值。
 - ❑ inc：为某个域值在原来的基础上增加或减小指定的数值，用于数字域。
- 这部分内容在 2.5.6 章节也有提及，你也可以翻到该章节重新回顾学习。



注意 用于原子更新的域必须是存储域，即 `stored="true"`。

使用 SolrJ 实现原子更新的示例代码，如下所示：

```
/**
 * Created by Lanxiaowei
 * SolrJ 原子更新测试
```

```

*/
public class TestAutomaticUpdate {
    private static final String SOLR_INSTANT_CORE = "http://localhost:8080/solr/
mapreduce";

    public static void main(String[] args) throws Exception {
        SolrClient client = new HttpSolrClient(SOLR_INSTANT_CORE);
        SolrInputDocument doc = new SolrInputDocument();
        String id = UUID.randomUUID().toString();
        doc.addField("id", id);
        doc.addField("word", "Sqoop");
        doc.addField("count", 100);
        client.add(doc);
        // 采用软提交
        client.commit(false, false, true);
        // Real-time Get
        realTimeGet(id, client);
        // 原子更新
        SolrInputDocument `sdoc = new SolrInputDocument();
        sdoc.addField("id", id);
        Map<String, Object> fieldModifier1 = new HashMap<String, Object>(1);
        fieldModifier1.put("inc", 3);
        Map<String, Object> fieldModifier2 = new HashMap<String, Object>(1);
        fieldModifier2.put("remove", "Sqoop");
        sdoc.addField("count", fieldModifier1);           // count 值加 3
        sdoc.addField("word", fieldModifier2);           // 删除 word 域
        client.add(sdoc);
        // Real-time Get
        realTimeGet(id, client);
        // 最终释放资源
        client.close();
    }

    private static void realTimeGet(String id, SolrClient client) throws Exception {
        SolrQuery query = new SolrQuery();
        // Real time GET 实时获取我们刚刚软提交的那个索引文档
        query.setRequestHandler("/get");
        query.set("id", id);
        query.set("fl", "id,word,count");
        QueryResponse response = client.query(query);
        System.out.println(" 以下是 Real time GET 的响应信息: \n");
        System.out.println(response.toString());
    }
}

```

上面示例代码中涉及的域的 schema 定义如下所示:

```

<field name="id" type="string" indexed="true" stored="true" />
<field name="_version_" type="long" indexed="true" stored="true"/>
<field name="word" type="string" indexed="true" stored="true" docValues="true" />
<field name="count" type="int" indexed="true" stored="true" docValues="true" />

```

当你执行原子更新时, 需要考虑并发问题, 当更新某个域的域值之前, 你要确保原子

更新操作的并发性问题，比如你先读取某个域的域值为 100，然后你想将该域的域值加 10，此时另外一个用户将该域的域值减 10 变成 90，如果没有并发乐观锁，那么你的文档更新操作会覆盖其他用户的操作，这样就会造成索引文档更新操作冲突，即多个索引文档更新应该以哪个为准。为了解决这类问题，Solr 提供了并发乐观锁控制功能，这个功能是通过一个 `_version_` 域来实现的，即每次更新都产生一个新的版本，你提交更新时，需要比对你提交的索引文档的版本号是否与当前索引文档的最新版本号相等，如果不相等，则会返回 Http 409 状态码，提示你索引文档的版本号冲突，更新失败。

Solr 默认自动为所有索引文档都添加了 `_version_` 域，Solr 客户端可以显式地为每个索引更新请求指定一个 `_version_` 版本值来实现并发乐观锁控制。表 13-2 总结了 Solr 更新请求中指定各种 `_version_` 值表示的语义。

表 13-2 `_version_` 值语义

<code>_version_</code>	语 义
<code>> 1</code>	索引文档的版本号必须精确匹配
<code>1</code>	索引文档必须存在
<code>< 0</code>	索引文档必须不存在
<code>0</code>	不受并发乐观锁控制，会存在并发更新问题

下面将以一个具体的示例来演示 Solr 中并发乐观锁的使用，首先请创建一个名称为 "optimistic-concurrency" 的测试 Core（具体请查阅随书源码），并按照如下配置示例定义 `schema.xml`：

```
<field name="id" type="string" indexed="true" stored="true" />
<field name="title" type="string" indexed="true" stored="true" />
<field name="author" type="string" indexed="true" stored="true" />
<field name="borrowOut" type="int" indexed="true" stored="true" />
<field name="_version_" type="long" indexed="true" stored="true" />
```

然后我们先添加一个测试文档，示例代码如下所示：

```
private static String addDoc(SolrClient client) throws Exception {
    SolrInputDocument doc = new SolrInputDocument();
    String id = UUID.randomUUID().toString();
    doc.addField("id", id);
    doc.addField("title", "book1");
    doc.addField("author", "Mr Hunter");
    doc.addField("borrowOut", 999);
    client.add(doc);
    // 采用软提交
    client.commit(false, false, true);
    return id;
}
```

然后我们采用多线程并发的方式来原子更新索引文档的 `title` 域的值，在每个线程原子更新完成之后实时 GET 当前最新的 `title` 域的值，示例代码如下所示：

```

// 线程池
private static ExecutorService executor = Executors.newFixedThreadPool(3);
public static void main(String[] args) throws Exception {
    SolrClient client = new HttpSolrClient(SOLR_INSTANT_CORE);
    // 先添加一个测试文档
    String id = addDoc(client);
    int taskCount = 5;
    for(int i=0; i < taskCount; i++) {
        FutureTask<Integer> task = new FutureTask<Integer>(new MyCallable(i,id,client));
        executor.submit(task);
    }
}

private static void atomicUpdate(int threadno,String id,SolrClient client) throws
Exception {
    String[] title = {"test1","test2","test3","test4","test5","test6","test7","te
st8","test9","test10"};
    // 原子更新
    SolrInputDocument sdoc = new SolrInputDocument();
    sdoc.addField("id",id);
    Map<String,Object> fieldModifier = new HashMap<String,Object>(1);
    String titleRanom = title[GernerlUtils.generateRandomNumber(9,0)];
    System.out.println("threadno:" + threadno + "/titleRanom:" + titleRanom);
    fieldModifier.put("set",titleRanom);
    sdoc.addField("title", fieldModifier); // borrowOut 值加1
    client.add(sdoc);
}

private static int realTimeGet(int threadno,String id,SolrClient client) throws
Exception {
    SolrQuery query = new SolrQuery();
    // Real time GET 实时获取当前最新版本的索引文档
    query.setRequestHandler("/get");
    query.set("id", id);
    query.set("fl", "id,title,author,borrowOut,_version_");
    QueryResponse response = client.query(query);
    NamedList<Object> result = response.getResponse();
    System.out.println("/*****");
    System.out.println(" 以下是 [" + threadno + "]Real time GET 的响应信息:");
    SolrDocument solrDocument = (SolrDocument) result.get("doc");
    String idVal = solrDocument.getFieldValue("id").toString();
    String title = solrDocument.getFieldValue("title").toString();
    String author = solrDocument.getFieldValue("author").toString();
    int borrowOut = Integer.valueOf(solrDocument.getFieldValue("borrowOut").toString());
    long version = Long.valueOf(solrDocument.getFieldValue("_version_").toString());
    System.out.println("id: " + idVal);
    System.out.println("title: " + title);
    System.out.println("author: " + author);
    System.out.println("borrowOut: " + borrowOut);
    System.out.println("_version_: " + version);
    System.out.println("/*****\n\n");
    return borrowOut;
}

```

```

static class MyCallable implements Callable<Integer> {
    private int threadno;
    private String id;
    private SolrClient client;
    public MyCallable(int threadno,String id,SolrClient client) {
        this.id = id;
        this.threadno = threadno;
        this.client = client;
    }
    public Integer call() throws Exception {
        // 原子更新
        atomicUpdate(threadno,id,client);
        // 然后实时 GET 最新值
        int borrowOut = realTimeGet(threadno,id,client);
        return borrowOut;
    }
}

```

通过测试，我们发现，每个线程获取到的值与我们实际预想的并不一样，难道 Solr 的并发乐观锁没起作用？之所以出现这种混乱的情况，是因为 Solr 并发乐观锁只保证在同一时刻永远只存在一个原子更新操作被执行，即在原子更新操作进行期间，不存在索引文档的域值被其他线程的原子更新操作所修改的情况发生。但是 Solr 乐观锁并不保证多个原子操作之间的并发性。比如示例中，我们先原子更新再实时 GET，这是两个原子操作，它们之间存在并发问题，即原子更新操作确实没有并发问题，能安全的更新，但是当实时 GET 最新索引文档且索引文档尚未返回之前，可能会有其他线程已经在此时将其更新，即你实时 GET 到的可能是已经被更新过的当前最新版本的索引文档，因此，示例中会出现每个线程实时 GET 获取到的结果是错乱的。这一点请务必知晓！想要解决这个并发问题，需要为原子更新和实时 GET 这两个操作加锁，保证这两个操作具有原子性。但是加锁后会有性能损耗，示例代码如下：

```

private static ReentrantLock lock = new ReentrantLock();
public Integer call() throws Exception {
    // 加锁
    lock.lock();
    atomicUpdate(threadno,id,client);
    int borrowOut = realTimeGet(threadno,id,client);
    // 解锁
    lock.unlock();
    return borrowOut;
}

```

13.9 使用 SolrJ 管理 Core

Solr 中提供了一套 Core HTTP 接口用于实现对 Core 的添加、加载、卸载、交换、重命

名等操作,关于 Core 的 HTTP 接口详细说明,请翻到 2.1.3 节进行查阅。使用 SolrJ 管理 Core 无非就是发送 HTTP 请求至 Solr Server,通过请求参数来指示 Solr Server 进行相关的 Core 操作,这里的 Core 操作类型是由 action 参数决定的,比如 action=create 即表示指示 Solr Server 创建一个新 Core。在 Solr 中所有的 Core 请求都使用 CoreAdminRequest 类来表示,如图 13-4 所示,下面将逐一介绍 Solr 中如何通过 CoreAdminRequest 来完成 Core 的常见操作。

13.9.1 创建 Core

每个 Core 创建之前都需要提前创建好 Core 根目录即 instanceDir,然后默认 Solr 会在 instanceDir 目录下搜寻 core.properties 配置文件,根据 core.properties (关于 core.properties 配置文件的详细配置说明,请翻到 3.1 章节重新回顾)中配置参数来确定数据目录 dataDir 以及配置目录 conf、schema.xml 和 solrconfig.xml 文件的名称。如果 core.properties 配置文件不存在,那么 Solr 默认就排除该

Core 不自动加载。但是对于采用 SolrJ 方式创建 Core 的情况下,Solr 则会根据用户传入的参数来决定配置文件的加载路径,不会考虑 core.properties 配置文件。如果用户未传递该参数,那么会采用默认值。比如 dataDir 的默认值是 data 即当前 Core 下的 data 目录,同理配置目录默认值是 conf, schema 和 solrconfig 的默认文件名称就是 schema.xml 和 solrconfig.xml。也就是说创建一个 Core 最简单的方式,你只需要创建一个 Core 目录,然后在 Core 目录下创建 data 和 conf 目录,并在 conf 目录下创建 schema.xml 和 solrconfig.xml。schema.xml 和 solrconfig.xml 这两个配置文件是创建 Core 的关键,两者其中任意一个加载失败都会导致 Core 创建失败。当然,如果需要自定义为其他值,那么你需要显式设置。关于创建 Core 有哪些可以传递的参数,请翻到 2.1.3 节做详细了解。使用 SolrJ 创建 Core 的示例代码,如下所示:

```
private static final String SOLR_URL = "http://localhost:8080/solr/";
SolrClient client = new HttpSolrClient(SOLR_URL);
```

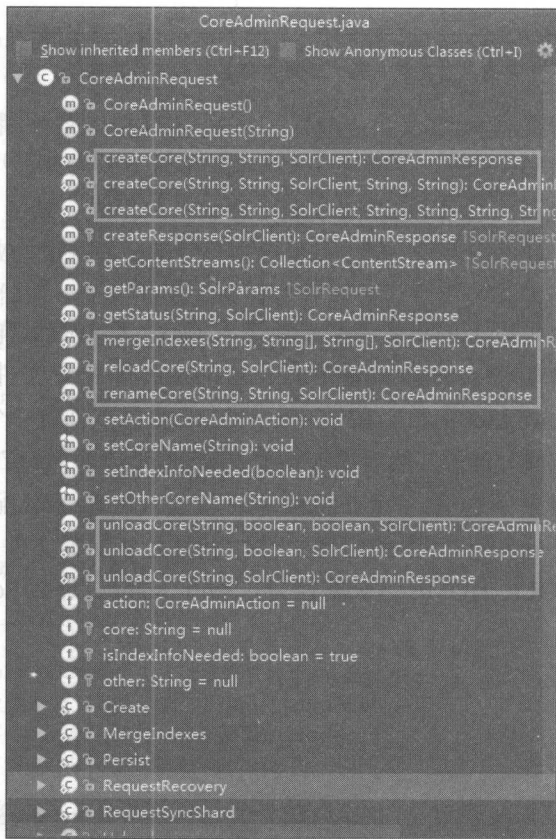


图 13-4 CoreAdminRequest 类提供的 Core 操作工具方法

```

Create create = new Create();
create.setPath("/admin/cores");
create.setCoreName("test");
create.setInstanceDir("C:/solr_home/test");
// 这些都是默认值, 如果你需要自定义为其他值, 那么你需要显式设置
// create.setDataDir("data");
// create.setSchemaName("schema.xml");
// create.setConfigName("solrconfig.xml");
// create.setIsLoadOnStartup(true);
// create.setIsTransient(false);
NamedList<Object> result = client.request(create);
System.out.println(result);

```

以上的设置参数表示含义在 3.1 章节有做详细解释, 这里就略过了。有时候, 你可能希望将 schema.xml 和 solrconfig.xml 等配置文件在多个 Core 之间共享, 此时你可以设置 configSet 参数。在 SOLR_HOME 目录下创建一个 configsets (这个目录名称是固定, 不能自定义) 目录, 然后添加多个 configSet 目录 (configSet 目录名称可以自定义), 每个 configSet 目录下包含一套 schema.xml 和 solrconfig.xml 配置文件, 然后多个 Core 之间可以共享一个 configSet 下的配置文件, 引用共享配置文件只需要根据 configSet 目录名称即可。假设 SOLR_HOME 目录路径为 C:/solr_home, 那么 Solr 的配置文件共享目录结构应该如下所示:

```

C:/solr_home
--configSets( 固定名称 )
  --testConfigSet1( 这个名称可以自定义 )
    --conf( 固定名称 )
      --schema.xml
      --solrconfig.xml
  --testConfigSet2( 这个名称可以自定义 )
    --conf( 固定名称 )
      --schema.xml
      --solrconfig.xml

```

这样你在创建 Core 的时候, 就不需要在当前 Core 目录的 conf 目录下存放 schema.xml 和 solrconfig.xml, 直接通过 configSet 参数设置引用共享配置目录下哪个 configSet 即可, Solr 会自动在指定 configSet 目录下查找需要的 schema.xml 和 solrconfig.xml 配置文件, 当然其他配置文件也可以放到共享目录下, 比如停用词字典、同义词字典等。

```

create.setInstanceDir("C:/solr_home/test");
create.setConfigSet("testConfigSet"); // 设置 configSet, 从共享配置目录下加载配置文件

```

13.9.2 卸载 Core

卸载一个 Core 只需要使用 CoreAdminRequest 类内部的 Unload 静态类, 设置需要卸载的 core 名称即可, 使用 SolrJ 卸载 Core 的示例代码, 如下所示:

```

SolrClient client = new HttpSolrClient(SOLR_URL);

```

```
CoreAdminRequest.Unload unload = new CoreAdminRequest.Unload(false);
unload.setPath("/admin/cores");
unload.setCoreName("test");
NamedList<Object> result = client.request(unload);
System.out.println(result);
```

注意，默认卸载 Core 并不会真正去删除硬盘上的 Core 目录，只是不再受 Core 容器管理罢了，你可以随时再重新加载它。当然，如果你确实希望能够彻底删除硬盘上的 Core 目录，那么你需要额外传递 `deleteInstanceDir` 参数，如果只是希望删除 Core 目录下的 `data` 目录，那么你需要额外传递 `deleteDataDir` 参数，如果你仅仅只是希望删除 Core 目录下的 `data/index` 目录下的索引数据，那么你需要额外传递 `deleteIndex`，有关这方面参数更详细说明请翻到 2.1.3 章节。体现到 SolrJ 上就是下面这 3 个方法的设置，示例代码如下所示：

```
// unload.setDeleteInstanceDir(true); // 删除整个 Core 目录
// unload.setDeleteDataDir(true); // 删除 Core 的 data 目录
// unload.setDeleteIndex(true); // 删除 Core 目录下的索引数据
```

13.9.3 加载 Core

加载指定 Core 需要传递 `action`、`core` 两个参数且 `action=LOAD`，`persist` 参数为可选。使用 SolrJ 加载指定 Core 的示例代码，如下所示：

```
CoreAdminRequest request = new CoreAdminRequest();
request.setCoreName("test");
request.setAction(CoreAdminParams.CoreAdminAction.LOAD);
NamedList<Object> result = client.request(request);
```

同理还有重新加载 Core，此时只需要将 `action` 参数值修改为 `RELOAD`。不管是 `load` 还是 `reload`，前提都是该 Core 必须已经被创建，如果该 Core 被卸载了，那么此时进行 `load` 或者 `reload` 将会提示 Core 不存在。

13.9.4 交换 Core

当在某个 Core 上执行全量更新，更新完毕后，你可能希望将线上的 Core 替换为全量更新后的 Core。此时你需要 Solr 的 Core 交换功能，Core 交换需要传递 3 个参数：`action`、`core`、`other` 参数，其中 `action` 参数必须为 `SWAP`。使用 SolrJ 实现 Core 交换的示例代码，如下所示：

```
CoreAdminRequest swap = new CoreAdminRequest();
// 将 "test" Core 与 "test2" Core 交换
swap.setCoreName("test");
swap.setOtherCoreName("test2");
swap.setAction(CoreAdminParams.CoreAdminAction.SWAP);
NamedList<Object> result = client.request(swap);
System.out.println(result);
```

13.9.5 重命名 Core

Core 重命名和 Core 交换需要传递的参数是相同的，只是此时 action 必须为 RENAME，且 other 参数表示重命名后的 Core 新名称。是使用 SolrJ 实现 Core 重命名的示例代码，如下所示：

```
CoreAdminRequest rename = new CoreAdminRequest();
// 将 "test" Core 重命名为 "test3"
rename.setCoreName("test");
rename.setOtherCoreName("test3");
rename.setAction(CoreAdminParams.CoreAdminAction.RENAME);
NamedList<Object> result = client.request(rename);
```

13.9.6 查看 Core 状态

获取指定 core 的运行状态信息需要指定 2 个参数：action 和 core，此时 action 参数值为 STATUS，若没有指定 core 参数，即表示返回所有 core 的运行状态。使用 SolrJ 返回 Core 运行状态的示例代码，如下所示：

```
CoreAdminResponse response = CoreAdminRequest.getStatus("core2", client);
NamedList<Object> result = response.getResponse();
SimpleOrderedMap map = (SimpleOrderedMap) result.get("status");
map = (SimpleOrderedMap) map.get("core2");
for (Object obj : map) {
    Map.Entry entry = (Map.Entry) obj;
    System.out.println(entry.getKey() + " : " + entry.getValue());
}
```

使用 SolrJ 返回所有 Core 运行状态的示例代码，如下所示：

```
CoreAdminRequest request = new CoreAdminRequest();
request.setAction(CoreAdminParams.CoreAdminAction.STATUS);
NamedList<Object> result = client.request(request);
System.out.println(result);
SimpleOrderedMap map = (SimpleOrderedMap) result.get("status");
for (Object obj : map) {
    Map.Entry entry = (Map.Entry) obj;
    SimpleOrderedMap coreMap = (SimpleOrderedMap) entry.getValue();
    for (Object object : coreMap) {
        Map.Entry entry2 = (Map.Entry) object;
        System.out.println(entry2.getKey() + " : " + entry2.getValue());
    }
    System.out.println();
}
```

13.9.7 Core 合并

合并 Core 索引即多个 Core 的索引数据合并到一个 Core 中，或者将多个索引目录合并到一个索引目录，后者不要求索引目录存在于 Core 中。前者需要传递多个 srcCore 参数

表示待合并的 Core 名称, 后者需要传递多个 indexDir 参数, 表示多个索引目录路径。此时 action 参数值为 mergeindexes。使用 SolrJ 实现多个 Core 索引目录合并的示例代码, 如下所示:

```
CoreAdminRequest.MergeIndexes mergeIndex = new CoreAdminRequest.MergeIndexes();
mergeIndex.setCoreName("test1");
String[] srcCores = new String[] {"test2","test3"};
mergeIndex.setSrcCores(Arrays.asList(srcCores));
NamedList<Object> result = client.request(mergeIndex);
```

至于如何使用 SolrJ 合并多个索引目录至指定 Core 中, 这里就不再演示了, 其实差不多, 你只需要将 setSrcCores 方法替换为 setIndexDirs 方法即可, 完整源码请查阅随书源码。

13.9.8 Core 分裂

同理还有 Core 分裂即将指定 Core 分裂成多个 Core, 或者将指定的索引目录分裂成多个索引目录, 这个 Core 操作与 Core 索引目录合并很类似, 仅仅是参数名称不同, 用法完全雷同, 此时 action 参数值为 split。不过可惜的是, SolrJ 并没有提供 split 操作的方法实现, 想要通过 SolrJ 完成 split 操作必须自己扩展, 以下是我扩展的 SplitIndexes 类用于实现 Core 分裂操作:

```
/**
 * Created by Lanxiaowei
 * Core 分裂或索引目录分裂
 */
public class SplitIndexes extends CoreAdminRequest {
    protected List<String> paths;
    protected List<String> targetCores;
    public static final String PATH_PARAM_NAME = "path";
    public SplitIndexes() {
        action = CoreAdminParams.CoreAdminAction.SPLIT;
    }
    public void setPaths(List<String> paths) {
        this.paths = paths;
    }
    public List<String> getPaths() {
        return paths;
    }
    public List<String> getTargetCores() {
        return this.targetCores;
    }
    public void setTargetCores(List<String> targetCores) {
        this.targetCores = targetCores;
    }
    @Override
    public SolrParams getParams() {
        if (action == null) {
```

```

        throw new RuntimeException("no action specified!");
    }
    ModifiableSolrParams params = new ModifiableSolrParams();
    params.set(CoreAdminParams.ACTION, action.toString());
    params.set(CoreAdminParams.CORE, core);
    if (paths != null) {
        for (String path : paths) {
            params.add(PATH_PARAM_NAME, path);
        }
    }
    if (targetCores != null) {
        for (String targetCore : targetCores) {
            params.add(CoreAdminParams.TARGET_CORE, targetCore);
        }
    }
    return params;
}
}

```

然后你需要使用自定义的 `SplitIndexes` 来发送 `split` 请求，跟上一章节使用 `MergeIndexes` 完全雷同。关于如何使用刚刚自定义的 `SplitIndexes` 来实现 `Core` 分裂或者索引目录分裂，请读者自行查阅随书源码，里面提供了完整的示例源码，这里为了节省篇幅就不演示了。

13.10 使用 SolrJ 管理 schema.xml

SolrJ 中的 `schema.xml` 管理主要是通过访问 `SchemaRequest` 类来实现，`SchemaRequest` 类向外部暴露的 `Schema API` 提供了大量关于 `schema` 的管理功能，比如域管理、域类型管理、`UniqueKey` 管理、全局 `Similarity` 管理、`Schema` 版本号管理等。但是，使用 Solr 中的 `Schema` 动态管理有一个前提，需要在 `solrconfig.xml` 中配置 `ManagedIndexSchemaFactory`。关于 `ManagedIndexSchemaFactory` 如何配置以及 Solr 的 `Schema API` 的详细介绍，请翻到 3.3.7 章节重新回顾。

13.10.1 Field 管理

每一种 `schema` 管理相关操作基本都对应着 `SchemaRequest` 类内部定义的一个静态类，跟 `Field` 管理相关的有：`AddCopyField`、`DeleteCopyField`、`AddField`、`ReplaceField`、`DeleteField`、`Field`、`Fields` 等，下面将逐一介绍每个类如何使用。

1. 普通 Field 管理

(1) 添加普通 Field

添加普通域需要使用 `AddField` 来完成，域定义相关信息通过 `Map` 类型来装载，在 `AddField` 类的构造函数中传入该 `Map`，然后调用 `AddField` 的 `process` 方法即可实现动态添加一个普通

域，下面是使用 SolrJ 动态添加一个普通 Field 的示例代码，如下所示：

```
Map<String, Object> fieldAttributes = new HashMap<String, Object>();
fieldAttributes.put("name", "product_name");
fieldAttributes.put("type", "string");
fieldAttributes.put("stored", true);
SchemaRequest.AddField addField = new SchemaRequest.AddField(fieldAttributes);
SchemaResponse.UpdateResponse response = addField.process(client, CORE_NAME);
```

(2) 更新普通 Field

当你通过 AddField 动态添加了一个普通域，你可能期望修改下域名称或者域类型或者域的其他属性等，此时你需要使用 ReplaceField 来完成，用法与 AddField 类基本雷同，使用 SolrJ 动态更新一个普通 Field 信息的示例代码，如下所示：

```
Map<String, Object> fieldAttributes = new HashMap<String, Object>();
fieldAttributes.put("name", "product_name");
fieldAttributes.put("type", "date");
fieldAttributes.put("stored", true);
fieldAttributes.put("omitNorms", true);
SchemaRequest.ReplaceField replaceField = new SchemaRequest.ReplaceField(
    fieldAttributes);
SchemaResponse.UpdateResponse response = replaceField.process(client, CORE_
NAME);
```

(3) 删除普通 Field

有时候，你可能希望能够动态删除某个普通域，那么你需要使用 DeleteField 类来完成，此时你只需要传入一个 name 参数表示你想要删除的域名称，使用 SolrJ 动态删除一个普通 Field 的示例代码，如下所示：

```
String fieldName = "product_name";
SchemaRequest.DeleteField deleteField = new SchemaRequest.DeleteField(fieldName);
SchemaResponse.UpdateResponse response = deleteField.process(client, CORE_
NAME);
```

(4) 查看 Field 的定义信息

如果想要查看指定域的定义信息，那么你需要使用 SchemaRequest 类的内部静态类 Field 来完成，在构造该内部静态类 Field 时，你只需要传入一个 fieldName 参数即可，使用 SolrJ 返回指定域的定义信息的示例代码，如下所示：

```
String fieldName = "name";
SchemaRequest.Field field = new SchemaRequest.Field(fieldName);
SchemaResponse.FieldResponse response = field.process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();
result = (NamedList<Object>) result.get("field");
for (Map.Entry<String, Object> entry : result) {
    String key = entry.getKey();
    Object val = entry.getValue();
```

```

        System.out.println(key + ": " + (val == null? "" : val.toString()));
    }
}

```

(5) 查看所有 Field 的定义信息

查看 schema.xml 中定义的所有域信息需要使用 SchemaRequest 类的内部静态类 Fields 来完成, 在构造该静态类时默认你可以不需要传递任何参数, 但是你也可以选择性的设置 fl、includeDynamic、showDefaults 参数来控制返回哪些域定义信息。使用 SolrJ 返回 schema.xml 中所有域的定义信息的示例代码, 如下所示:

```

ModifiableSolrParams params = new ModifiableSolrParams();
// 是否显示域类型的默认属性信息
params.add("showDefaults", "true");
// 是否返回动态域的定义信息
params.add("includeDynamic", "true");
// 指定返回哪些域的定义信息
params.add("fl", "id,name,product_name,_version_");
SchemaRequest.Fields allFields = new SchemaRequest.Fields(params);
SchemaResponse.FieldsResponse response = allFields.process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();
List<NamedList<Object>> fields = (List<NamedList<Object>>)result.get("fields");
for(NamedList<Object> field : fields) {
    for(Map.Entry<String,Object> entry : field) {
        String key = entry.getKey();
        Object val = entry.getValue();
        System.out.println(key + ": " + (val == null? "" : val.toString()));
    }
}
}

```

2. DynamicField 管理

(1) 添加 DynamicField

添加 DynamicField (即动态域) 需要使用 AddDynamicField 类完成, 它与添加普通域的 AddField 类的用法基本雷同, 此时的 name 参数值需要使用通配符, 比如 "*_s"。关于如何使用 AddDynamicField 动态添加一个动态域, 可以参考添加普通 Field 的示例代码, 或者查阅随书源码提供的示例, 这里略过。

(2) 更新 DynamicField

更新动态域需要使用 ReplaceDynamicField 类, 用法与 ReplaceField 类基本雷同, 具体如何使用 ReplaceDynamicField 类实现动态更新一个动态域的信息, 请查阅随书源码, 这里略过。

(3) 删除 DynamicField

删除一个动态域需要使用 DeleteDynamicField 类完成, 用法与 DeleteField 类基本雷同, 具体如何使用 ReplaceDynamicField 类实现动态删除一个动态域, 请查阅随书源码, 这里略过。

(4) 查看 DynamicField 的定义信息

查看 schema.xml 中定义的所有动态域信息需要使用 SchemaRequest 类的内部静态类 DynamicField，用法与静态类 Field 基本类似，使用 SolrJ 返回指定动态域的定义信息的示例代码，如下所示：

```
String fieldName = "*_n";
SchemaRequest.DynamicField dynamicField = new SchemaRequest.DynamicField(fieldName);
SchemaResponse.DynamicFieldResponse response = dynamicField.process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();
System.out.println(result);
result = (NamedList<Object>) result.get("dynamicField");
for(Map.Entry<String, Object> entry : result) {
    String key = entry.getKey();
    Object val = entry.getValue();
    System.out.println(key + ": " + (val == null? "" : val.toString()));
}
```

(5) 查询所有 DynamicField 的定义信息

查看 schema.xml 中定义的所有动态域信息需要使用 SchemaRequest 类的内部静态类 DynamicFields，用法与静态类 Fields 基本类似，使用 SolrJ 返回 schema.xml 中所有动态域的定义信息的示例代码，如下所示：

```
SchemaRequest.DynamicFields allDynamicFields = new SchemaRequest.DynamicFields();
SchemaResponse.DynamicFieldsResponse response = allDynamicFields.process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();
List<NamedList<Object>> dynamicFields = (List<NamedList<Object>>)result.get("dynamicFields");
for(NamedList<Object> dynamicField : dynamicFields) {
    for(Map.Entry<String, Object> entry : dynamicField) {
        String key = entry.getKey();
        Object val = entry.getValue();
        System.out.println(key + ": " + (val == null? "" : val.toString()));
    }
}
```

3. CopyField 管理

(1) 添加 CopyField

添加一个 CopyField（即复制域）需要使用 AddCopyField 类，构造 AddCopyField 需要传入 source 和 dest 两个参数，这里 dest 参数设计为 List 类型，主要目的是为了实现将一个域批量复制到多个目标域上，是使用 SolrJ 实现动态添加 CopyField 的示例代码，如下所示：

```
String sourceField = "name";
List<String> destFields = new ArrayList<String>();
destFields.add("product_name");
```

```

destFields.add("title");
SchemaRequest.AddCopyField addCopyField = new SchemaRequest.AddCopyField(sourceField, destFields);
SchemaResponse.UpdateResponse response = addCopyField.process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();

```



注意 CopyField 没有更新操作。

(2) 删除 CopyField

删除一个 CopyField 需要使用 DeleteCopyField 类完成，在构造该类时，你需要传入 sourceFieldName 和 destFieldName 两个参数，其中 destFieldName 参数被设计为 List 类型，是为了实现批量删除，使用 SolrJ 实现动态删除 CopyField 的示例代码，如下所示：

```

String sourceField = "name";
String destField = "title";
List<String> destFields = new ArrayList<String>();
destFields.add(destField);
SchemaRequest.DeleteCopyField deleteCopyField = new SchemaRequest.DeleteCopyField(sourceField, destFields);
SchemaResponse.UpdateResponse response = deleteCopyField.process(client, CORE_NAME);

```

(3) 查看 CopyField 的定义信息

查看 schema.xml 中定义的所有复制域信息需要使用 SchemaRequest 类的内部静态类 CopyFields，用法与静态类 DynamicFields 基本类似。还可以额外指定 source.fl 和 dest.fl 参数来限制返回哪些 Source Field 和 Dest Field 的定义信息。使用 SolrJ 返回 schema.xml 中所有复制域的定义信息的示例代码，如下所示：

```

String fieldName = "name";
ModifiableSolrParams params = new ModifiableSolrParams();
params.add("wt", "json");
// 设置返回哪些 source field 信息
params.add("source.fl", "name");
// 设置返回哪些 dest field 信息
params.add("dest.fl", "product_name, title");
SchemaRequest.CopyFields allCopyFields = new SchemaRequest.CopyFields(params);
SchemaResponse.CopyFieldsResponse response = allCopyFields.process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();
List<NamedList<Object>> copyFields = (List<NamedList<Object>>) result.get("copyFields");
for(NamedList<Object> copyField : copyFields) {
    for(Map.Entry<String, Object> entry : copyField) {
        String key = entry.getKey();

```

```

Object val = entry.getValue();
System.out.println(key + ": " + (val == null? "" : val.toString()));
}
}

```

13.10.2 FieldType 管理

每个 Field 都对应着一个 FieldType，除了能够动态添加删除更新 Field，还可以动态的添加删除更新 FieldType。SolrJ 中是通过 SchemaRequest 类内部的各种静态类来完成对 FieldType 的动态管理，主要涉及的静态类有：AddFieldType、ReplaceFieldType、DeleteFieldType、FieldType、FieldTypes。下面将逐一介绍如何使用每个静态内部类来完成对 schema.xml 中 FieldType 的动态管理。

1. 添加 FieldType

想要动态往 schema.xml 中添加域类型，需要使用 SchemaRequest 类的内部静态类 AddFieldType 类来完成，构造 AddFieldType 类时，需要传入一个 FieldTypeDefinition 对象，而构造一个 FieldTypeDefinition 对象，需要设置如下图 13-5 所示的几个属性，这几个属性分别对应着 < fieldType> 的属性以及其下级元素：

```

public class FieldTypeDefinition {
    private Map<String, Object> attributes;

    private AnalyzerDefinition analyzer;

    private AnalyzerDefinition indexAnalyzer;

    private AnalyzerDefinition queryAnalyzer;

    private AnalyzerDefinition multiTermAnalyzer;

    private Map<String, Object> similarity;
}

```

图 13-5 FieldTypeDefinition 类包含的属性

- attributes 属性：对应着 < fieldType> 元素自身的属性，比如 name、class、positionIncrementGap 等。
- analyzer 属性：对应着 < fieldType> 元素下面的 < analyzer> 子元素，当需要直接配置 Analyzer 实现类的 class 而不是配置 < tokenizer> 和 < filter> 时需要设置此属性。
- indexAnalyzer 属性：对应着 < fieldType> 元素下面的 < analyzer> 子元素，当需要配置索引时的分词器配置时需要设置此属性。
- queryAnalyzer 属性：对应着 < fieldType> 元素下面的 < analyzer> 子元素，当需要配置查询时的分词器配置时需要设置此属性。
- multiTermAnalyzer 属性：对应着 < fieldType> 元素下面的 < analyzer> 子元素，当需要应对类似前缀查询、正则查询这种特殊的 Multi-Term 查询并且期望不对用户输入的搜索关键字进行分词时你需要设置此属性。
- similarity 属性：对应着 < fieldType> 元素下面的 < similarity> 子元素，当需要为当前域类型配置 similarity 打分器时，你需要设置此属性。

以下将以动态添加一个 "text_general" 域类型为例演示如何使用 SolrJ 动态添加一个域类型，其中 "text_general" 域类型的定义如下所示：

```

< fieldType name="text_general" class="solr.TextField" positionIncrementGap="100">
  < analyzer type="index">

```

```

<tokenizer class="solr.StandardTokenizerFactory"/>
<filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt" />
<filter class="solr.LowerCaseFilterFactory"/>
</analyzer>
<analyzer type="query">
<tokenizer class="solr.StandardTokenizerFactory"/>
<filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt" />
<filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt" ignoreCase="true"
expand="true"/>
<filter class="solr.LowerCaseFilterFactory"/>
</analyzer>
</fieldType>

```

其中涉及的 stopwords.txt 和 synonyms.txt 两个字典文件，你需要提前放置到当前 Core 的 conf 目录下。

使用 SolrJ 动态添加 "text_general" 域类型的示例代码，如下所示：

```

Map<String, Object> fieldTypeAttributes = new HashMap<String, Object>();
fieldTypeAttributes.put("name", "text_general");
fieldTypeAttributes.put("class", "solr.TextField");
fieldTypeAttributes.put("positionIncrementGap", 100);
Map<String, Object> tokenizer = new HashMap<String, Object>();
tokenizer.put("class", "solr.StandardTokenizerFactory");
Map<String, Object> stopFilter = new HashMap<String, Object>();
stopFilter.put("class", "solr.StopFilterFactory");
stopFilter.put("ignoreCase", true);
stopFilter.put("words", "stopwords.txt");
Map<String, Object> lowerCaseFilter = new HashMap<String, Object>();
lowerCaseFilter.put("class", "solr.LowerCaseFilterFactory");
Map<String, Object> synonymFilter = new HashMap<String, Object>();
synonymFilter.put("class", "solr.SynonymFilterFactory");
synonymFilter.put("synonyms", "synonyms.txt");
synonymFilter.put("ignoreCase", true);
synonymFilter.put("expand", true);
AnalyzerDefinition indexAnalyzer = new AnalyzerDefinition();
Map<String, Object> indexAnalyzerAttributes = new HashMap<String, Object>();
indexAnalyzerAttributes.put("type", "index");
List<Map<String, Object>> indexFilters = new ArrayList<Map<String, Object>>();
indexFilters.add(stopFilter);
indexFilters.add(lowerCaseFilter);
indexAnalyzer.setAttributes(indexAnalyzerAttributes);
indexAnalyzer.setTokenizer(tokenizer);
indexAnalyzer.setFilters(indexFilters);
AnalyzerDefinition queryAnalyzer = new AnalyzerDefinition();
Map<String, Object> queryAnalyzerAttributes = new HashMap<String, Object>();
queryAnalyzerAttributes.put("type", "query");
List<Map<String, Object>> queryFilters = new ArrayList<Map<String, Object>>();
queryFilters.add(stopFilter);
queryFilters.add(synonymFilter);
queryFilters.add(lowerCaseFilter);

```



```

queryAnalyzer.setAttributes(queryAnalyzerAttributes);
queryAnalyzer.setTokenizer(tokenizer);
queryAnalyzer.setFilters(queryFilters);
FieldTypeDefinition fieldTypeDefinition = new FieldTypeDefinition();
fieldTypeDefinition.setAttributes(fieldTypeAttributes);
fieldTypeDefinition.setIndexAnalyzer(indexAnalyzer);
fieldTypeDefinition.setQueryAnalyzer(queryAnalyzer);
SchemaRequest.AddFieldType addFieldType = new SchemaRequest.AddFieldType(field
TypeDefinition);
SchemaResponse.UpdateResponse response = addFieldType.process(client,CORE_
NAME);
NamedList<Object> result = response.getResponse();

```

虽然功能实现了,但是我个人感觉编码量太大,略显繁琐,后续会讲解使用 JSON Request API 的方式来实现,该方法会显得更简洁。

2. 更新 FieldType

当通过 AddFieldType 类动态添加了一个 FieldType 之后,你可能期望重新对其进行修改,那么此时你需要使用 SchemaRequest 类的内部静态类 ReplaceFieldType, ReplaceFieldType 的用法与 AddFieldType 完全雷同,只需要将上面使用 SolrJ 动态添加 FieldType 的示例代码中的 AddFieldType 替换为 ReplaceFieldType 即可,这里就不再演示 ReplaceFieldType 如何使用了,更详细的示例代码请读者查阅随书源码。

3. 删除 FieldType

删除指定 FieldType 需要使用 SchemaRequest 类的内部静态类 DeleteFieldType,构造 DeleteFieldType 对象时传入一个域类型名称参数即可, DeleteFieldType 的用法与 DeleteField 几乎雷同,使用 SolrJ 动态删除一个域类型的示例代码,如下所示:

```

String fieldTypeName = "text_general";
SchemaRequest.DeleteFieldType deleteFieldType = new SchemaRequest.DeleteFieldType
(fieldTypeName);
SchemaResponse.UpdateResponse response = deleteFieldType.process(client,CORE_
NAME);
NamedList<Object> result = response.getResponse();

```

4. 查看指定 FieldType 的定义信息

同 Field 类似,也可以通过指定一个域类型名称参数来查看一个域类型的定义信息,此时你需要使用 SchemaRequest 类的内部静态类 FieldType 来完成。使用 SolrJ 实现查看指定 FieldType 的定义信息的示例代码,如下所示:

```

String fieldTypeName = "text_general";
SchemaRequest.FieldType fieldType = new SchemaRequest.FieldType(fieldTypeName);
SchemaResponse.FieldTypeResponse response = fieldType.process(client,CORE_
NAME);
NamedList<Object> result = response.getResponse();

```

```

result = (NamedList<Object>) result.get("fieldType");
for (Map.Entry<String, Object> entry : result) {
    String key = entry.getKey();
    Object val = entry.getValue();
    System.out.println(key + ": " + (val == null? "" : val.toString()));
}

```

5. 查看所有 FieldType 的定义信息

查看 schema.xml 中定义的所有域类型的定义信息你需要使用 SchemaRequest 类的内部静态类 FieldTypes 来完成, 在构造该静态类对象时你可以不需要传递任何参数, 使用 SolrJ 返回 schema.xml 中所有域类型的定义信息的示例代码, 如下所示:

```

ModifiableSolrParams params = new ModifiableSolrParams();
// 是否显示域类型的默认属性信息
params.add("showDefaults", "true");
SchemaRequest.FieldTypes allFieldTypes = new SchemaRequest.FieldTypes(params);
SchemaResponse.FieldTypesResponse response = allFieldTypes.process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();
List<NamedList<Object>> fieldTypes = (List<NamedList<Object>>) result.get("fieldTypes");
for (NamedList<Object> fieldType : fieldTypes) {
    for (Map.Entry<String, Object> entry : fieldType) {
        String key = entry.getKey();
        Object val = entry.getValue();
        System.out.println(key + ": " + (val == null? "" : val.toString()));
    }
    System.out.println("/*****");
}

```

13.10.3 Schema 管理

Schema 相关的管理操作主要是一些 schema 的名称、schema 的版本号、schema 中定义的 UniqueKey、Schema 中定义的全局 Similarity 等信息的查询。下面会逐一介绍如何使用 SolrJ 来获取 Schema 相关信息。

1. 查看 Schema 名称

如果想要查看 schema 的名称信息 (即 schema.xml 中 <schema> 元素的 name 属性值), 你需要使用 SchemaRequest 类的内部静态类 SchemaName。使用 SolrJ 获取 schema 名称信息的示例代码如下所示:

```

ModifiableSolrParams params = new ModifiableSolrParams();
params.add("wt", "json");
SchemaRequest.SchemaName schemaName = new SchemaRequest.SchemaName(params);
SchemaResponse.SchemaNameResponse response = schemaName.process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();
String name = result.get("name").toString();

```

2. 查看 Schema 版本号

同理还可以查看 schema 的版本信息（即 schema.xml 中 <schema> 元素的 version 属性值），你需要使用 SchemaRequest 类的内部静态类 SchemaVersion。使用 SolrJ 获取 schema 版本号的示例代码如下所示：

```
SchemaRequest.SchemaVersion schemaVersion = new SchemaRequest.SchemaVersion();
SchemaResponse.SchemaVersionResponse response = schemaVersion.process(client, CORE_
NAME);
NamedList<Object> result = response.getResponse();
String version = result.get("version").toString();
```

3. 查看 Schema 中定义的 UniqueKey

你还可以查看 Schema 中定义的 uniqueKey 信息，此时你需要使用 SchemaRequest 类的内部静态类 UniqueKey。使用 SolrJ 获取 schema.xml 中定义的 UniqueKey 信息的示例代码如下所示：

```
SchemaRequest.UniqueKey uniqueKey = new SchemaRequest.UniqueKey();
SchemaResponse.UniqueKeyResponse response = uniqueKey.process(client, CORE_
NAME);
NamedList<Object> result = response.getResponse();
String uniquekey = result.get("uniqueKey").toString();
```

4. 查看整个 Schema 信息

当然你也可以直接查看整个 schema.xml 中定义的所有信息，此时你需要使用 SchemaRequest 类来完成。使用 SolrJ 获取整个 schema.xml 中定义的信息的示例代码如下所示：

```
SchemaRequest request = new SchemaRequest();
request.setPath("/schema");
SchemaResponse response = request.process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();
result = (NamedList<Object>)result.get("schema");
String schemaName = result.get("name").toString();
String version = result.get("version").toString();
List<NamedList<Object>> fieldTypes = (List<NamedList<Object>>)result.get("field
Types");
List<NamedList<Object>> fields = (List<NamedList<Object>>)result.get("fields");
```

5. 查看 Schema 中定义的全局 Similarity 信息

你还可以通过 SchemaRequest 类内部的静态类 GlobalSimilarity 来返回 schema.xml 中定义的全局 Similarity 信息，至于如何获取某个域中定义的 Similarity 信息，请翻到 13.10.2 节进行了解。使用 SolrJ 实现获取 schema.xml 中定义的全局 Similarity 信息的示例代码如下所示：

```
SchemaRequest.GlobalSimilarity globalSimilarity = new SchemaRequest.
GlobalSimilarity();
SchemaResponse.GlobalSimilarityResponse response = globalSimilarity.
```

```
process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();
result = (NamedList<Object>) result.get("similarity");
String similarityClass = result.get("class").toString();
```

6. 查看 Schema 中定义的查询连接符

你还可以 Solr 提供的 Schema API 获取到 schema.xml 中定义的 defaultQueryOperator 信息, 如果 schema.xml 中没有显式定义 defaultQueryOperator, 那么 Solr 会返回 QueryOperator 的默认值。在 SolrJ 中返回 schema.xml 中定义的 defaultQueryOperator 信息, 需要使用 SchemaRequest 类内部的静态类 DefaultQueryOperator 来完成。使用 SolrJ 来实现返回 schema.xml 中定义的 defaultQueryOperator 信息的示例代码如下所示:

```
SchemaRequest.DefaultQueryOperator defaultQueryOperator = new SchemaRequest.
DefaultQueryOperator();
SchemaResponse.DefaultQueryOperatorResponse response = defaultQueryOperator.
process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();
String defaultOperator = result.get("defaultOperator").toString();
```

13.10.4 Schema 管理的事务性批量操作

以上介绍的都是对 Schema 的单个操作, 其实 Solr 的 Schema API 还提供了一种 Schema 批量管理操作并且还带有事务性质, 即多个 Schema 管理操作在单个请求里完成, 并且要么每个操作全部执行成功, 要么全部执行失败。在 SolrJ 实现这种批量操作需要使用 SchemaRequest 类内部的静态类 MultiUpdate 来完成, 而构造 MultiUpdate 对象需要传入 List<Update> 类型的参数, 其中 Update 表示 Schema 更新操作的抽象, 注意, 只有 Schema 的更新操作才支持批量操作, 也就意味着查询操作不支持事务性批量操作, 比如查看域或域类型的定义信息。使用 SolrJ 实现批量操作 Schema 的示例代码如下所示:

```
List<SchemaRequest.Update> updateSchemaRequests = new ArrayList<SchemaRequest.
Update>();
String fieldName = "book_name";
String fieldType = "text_it";
SchemaRequest.AddFieldType addFieldType = createAddFieldType(fieldTypeName);
SchemaRequest.AddField addField = createAddField(fieldName, fieldType);
SchemaRequest.DeleteField deleteField = new SchemaRequest.DeleteField(fieldName);
SchemaRequest.DeleteFieldType deleteFieldType = new SchemaRequest.DeleteFieldType
(fieldName);
updateSchemaRequests.add(addFieldType);
updateSchemaRequests.add(addField);
updateSchemaRequests.add(deleteField);
updateSchemaRequests.add(deleteFieldType);
SchemaRequest.MultiUpdate multiUpdate = new SchemaRequest.MultiUpdate(updateSc
hemaRequests);
SchemaResponse.UpdateResponse response = multiUpdate.process(client, CORE_
```

```
NAME);
NamedList<Object> result = response.getResponse();
```

以上的示例中，我们先创建了一个名称为 "text_it" 的域类型，然后创建了一个名称为 "book_name" 的域，并应用 "text_it" 域类型，最后依次删除 book_name 域和 text_it 域类型，这 4 个 Schema 管理操作在具有事务性的单个请求中完成，保证了 4 个操作的完整性和一致性。

13.11 使用 SolrJ 操作 JSON Request API

通常我们的查询参数都是手动去一个个指定的，但是这种方式有以下几个不足之处：

□ 参数无结构化，需要传入各种冗长又难看的参数名称，比如：

```
f.facet_name.facet.range.start=5
```

□ 参数值全部用 string 表示，基本无参数数据类型之分；

□ 当查询参数很多时，理解这个查询的含义变得困难；

□ 当查询参数很多时，不利于编程，设置参数的代码变得繁琐，这点我们已经在使用 SolrJ 动态添加域类型时已经感受到了；

□ 不能进行良好的参数验证。

为此，Solr 提供了全新的 JSON Request API 来解决此类问题，使用 JSON Request API 可以将请求参数采用 JSON 格式组织在一起，而不是像传统方式那样采用 key=value 将参数一个个分割开单独设置，这样即繁琐又不利于阅读和编码。

使用 JSON Request API 添加索引文档，此时可以使用 JSON 格式来描述索引文档，最后将 JSON 格式的索引文档数据当作 Request Payload 传递给 Solr Server，比如 JSON 格式的索引文档如下：

```
[{
  id: "1",
  product_name: "product-1",
  sell-count: 100
},
{
  id: "2",
  product_name: "product-2",
  sell-count: 200
}]
```

现在可以直接将这个 JSON 字符串进行创建索引，但是注意，即便只是添加一个索引文档，JSON 字符串两头还是需要有一对中括号 []。

对于之前的这种查询：

```
http://localhost:8080/solr/techproducts/select?q=memory&fq=inStock:true&wt=json
```

```
&indent=true
```

现在可以直接采用 JSON 格式来描述查询参数，格式如下：

```
{
  "query" : "memory",
  "filter" : "inStock:true"
}
```

然后可以采用 POST 方式将整个 JSON 字符串格式的参数当作一个 Request Payload 发送给 Solr Server，或者也可以采用 GET 请求将 JSON 参数以 key-value 形式进行传递，只不过此时的 key 为 json，value 为 JSON 字符串，如下所示：

```
http://localhost:8080/solr/techproducts/select?json={"query":"book_name:solr"}&wt=json&indent=true
```

当然还可以采用 GET 请求以 json.<path>=<json_value> 格式传递请求参数，比如：

```
http://localhost:8080/solr/techproducts/query?json.limit=5&json.filter="cat:electronics"&json.query="book_name:solr"
```

甚至可以将两者进行组合，如下所示：

```
http://localhost:8080/solr/techproducts/query?json.limit=5&json.filter="cat:electronics"
Request Payload:
{
  query : "memory",
  limit : 10
}
```

像那些通用的参数，比如 wt、indent、limit 等参数还可以使用 params 属性包裹起来，如下所示：

```
{
  params: {
    wt: json,
    indent: true,
  },
  rows: 10
},
query : "*:*",           // 等价于 q 参数
filter : [               // 等价于 fq 参数
  "author:brandon",
  "genre_s:fantasy"
],
offset : 0,              // 等价于 start 参数
limit : 5,               // 等价于 rows 参数
fields : ["title","author"], // 等价于 fl 参数，你也可以用字符串形式 "title,author"
sort : "sequence_i desc", // 你也可以写成 {"sequence_i":"desc","price":"asc"}
facet : {                // JSON Facet API 部分我们已经在 11.6 章节讲解过
```



```

    avg_price : "avg(price)",
    top_authors : {terms : author}
  }
}

```

关于 Facet 查询参数，还可以通过以 json.facet 为 key，JSON 格式的 JSON Facet 语法表达式为 val 的形式通过 GET 请求传递，示例如下：

```

json.facet={
  top_genres : {
    type : terms,
    field : genre_field,
    limit : 3,
    mincount : 2
  }
}

```

JSON Request API 的 JSON 格式参数中还支持以变量的形式引用外部 URL 中通过 key-value 形式定义请求参数，示例如下所示：

```

http://localhost:8080/solr/techproducts/query?FIELD=text&TERM=memory&HOWMANY=10"
// 这里是 request payload
{
  query: "${FIELD}:${TERM}",
  limit: ${HOWMANY}
}

```

对于 Core 和 Schema 管理相关 API 也同样支持 JSON Request API，比如添加一个域，请求参数用 JSON 格式表示如下所示：

```

{
  "add-field": {
    "name": "sell-by",
    "type": "tdate",
    "stored": true
  }
}

```

其他操作同理，具体请查阅 2.1.3 章节和 3.3.7 章节，这里不再重复说明。然而遗憾的是，SolrJ 对 JSON Request API 支持不太好（因为 JSON Request API 仍然属于实验性功能，尚不稳定），为此我对 SolrJ 进行了扩展，使其能够直接传递 JSON 格式的参数，以及能够直接解析 JSON 格式的响应数据。这里主要扩展了两个类：HttpJSONSolrClient 和 JSONResponseParser，扩展 HttpJSONSolrClient 主要是为了支持能够传递 Request Payload，扩展 JSONResponseParser 是为了支持返回 JSON，因为默认 SolrJ 是采用 javabin 格式，而我们传递的 Request Payload 是 JSON 格式，如果不进行扩展，会抛出 conent-type 类型不一致的异常。然而，SolrJ 到现在仍然没有提供 JSON 数据解析器，虽然扩展了 JSONResponseParser，但是无奈 Solr 返回的 JSON 数据的属性众多，需要完整解析并转换成 NameList 类型，牵一发而动全身，过于复杂，故直

接返回了 JSON 字符串，具体实现源代码请查阅随书源码。

使用扩展的 `HttpJSONSolrClient` 以 JSON 格式参数来动态添加域的示例代码，如下所示：

```
HttpJSONSolrClient client = new HttpJSONSolrClient(SOLR_URL);
String jsonParams =
    "{" +
    "  \"add-field\":{\"" +
    "    \"name\":\"sell-count4\"," +
    "    \"type\":\"int\"," +
    "    \"stored\":true" +
    "  }" +
    "}";
client.setJsonParams(jsonParams);
GenericSolrRequest addField = new
    GenericSolrRequest(SolrRequest.METHOD.POST, "/schema", null);
SimpleSolrResponse response = addField.process(client, CORE_NAME);
NamedList<Object> result = response.getResponse();
```

更多使用 SolrJ 操作 JSON Request API 的示例请查阅随书源码，这里就不一一列举了。关键还是需要大家对 Http 协议非常熟悉，比如知道什么是 Request Payload，然后需要熟悉 Solr 查询相关 API、Core 和 Schema 管理相关 API，按照 JSON Request API 传递参数的语法进行传递参数即可，如果实在不清楚的，请仔细阅读随书源码中提供的示例代码，尝试着自己将以往传统的查询写法，使用 SolrJ 以 JSON Request API 方式再实现一遍。

13.12 使用 Spring Data Solr

Spring Data Solr 是 Spring Data 家族的一份子，它基于 Spring IOC 和 SolrJ API 进行了一层薄封装，使得用户可以采用配置或者注解的方式来与 Solr Server 交互，而且也更容易将 Solr 集成到现有项目中。

13.12.1 Spring Data Solr 环境搭建

首先创建一个 Maven 项目，然后在 Maven 项目的 `pom.xml` 中添加 Spring Maven 仓库，如下所示：

```
<repositories>
<repository>
<id>spring-milestone</id>
<name>Spring Milestone Maven Repository</name>
<url>http://repo.springsource.org/libs-milestone</url>
</repository>
</repositories>
```

然后在 `pom.xml` 中添加 `spring-data-solr` 依赖，配置示例如下所示：

```

<dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-solr</artifactId>
<version>2.0.4.RELEASE</version>
</dependency>
<!-- 如果你需要使用 EmbeddedSolrServer, 那么你还需要添加下面的依赖 -->
<dependency>
<groupId>org.apache.solr</groupId>
<artifactId>solr-core</artifactId>
<version>5.3.1</version>
<exclusions>
<exclusion>
<artifactId>slf4j-jdk14</artifactId>
<groupId>org.slf4j</groupId>
</exclusion>
</exclusions>
</dependency>

```

然后在 classpath 路径下创建一个 application-solr.properties 属性文件, 在其中配置我们的 Solr Server 访问 URL, 如果你需要使用 EmbeddedSolrServer, 那么还需要配置 SOLR_HOME 参数, 配置示例如下所示:

```

solr.server.url=http://localhost:8080/solr/
solr.solr.home=C:/solr_home

```

接下来可以采用 Java Config 的方式来配置 Solr Client 实例, 示例代码如下所示:

```

@Configuration
@ComponentScan(basePackages = {"com.yida.solr.book.examples.ch13.springdatasolr.service"})
@EnableSolrRepositories(basePackages = {
    "com.yida.solr.book.examples.ch13.springdatasolr.repository"},
    multicoreSupport = true)
@PropertySource("classpath:application-solr.properties")
public class HttpSolrContext {
    @Bean
    public SolrClient solrClient(@Value("${solr.server.url}") String solrHost) {
        return new HttpSolrClient(solrHost);
    }
}

```

@EnableSolrRepositories 注解表示启用 Solr Repositories 包扫描, multicoreSupport 属性表示开启多 Core 支持。其实你也可以新建一个 applicationContext-solr.xml, 通过传统的 Spring 配置文件的形式进行配置, 配置示例如下所示:

```

<context:property-placeholder location="classpath:application-solr.properties"/>
<!-- 启用 Solr repositories 包扫描 -->
<solr:repositories base-package="com.yida.solr.book.examples.ch13.springdatasolr.repository"

```

```
repository-impl-postfix="Impl" multicore-support="true" />
<solr:solr-client id="solrClient" url="${solr.server.url}"/>
```

到此，Spring Data Solr 的开发环境就搭建完毕了。下面我们将学习如何使用 Spring Data Solr 创建 / 删除 / 更新 / 查询索引文档。

13.12.2 Spring Data Solr 的 CRUD

上一小节，我们学习了如何配置 Spring Data Solr，现在继续学习如何添加索引到 Solr Server，以及如何更新、删除、查询索引文档。

假设有一个 Core，Core 的 schema.xml 中有 id 和 product_name 两个域，配置如下：

```
<field name="id" type="string" indexed="true" required="true" stored="true"/>
<field name="name" type="string" stored="true"/>
```

然后需要使用 POJO 来表示它，并用 SolrJ 的注解将 POJO 的属性与 schema 中定义的域名称进行映射，下面创建一个名称为的 "Product" 的 POJO，示例代码如下所示：

```
/**
 * Created by Lanxiaowei
 * 产品类，用于描述 Schema 中定义的域并与之进行映射，从而便于以 OOP 的方式创建索引
 */
@SolrDocument(solrCoreName = "test2")
public class Product {
    @Id
    @Field
    private String id;
    @Field("name")
    private String productName;
    ...// get / set 方法以及构造方法省略
```

其中 @Field 注解是 SolrJ 提供的，用于类属性与域名称之间的映射，@Id 注解是 Spring Data Solr 提供的，用于标识这是一个 UniqueKey 域。solrCoreName 属性用于指定对应的 Core 名称

然后我们需要创建一个 Repository 接口（可以将 Repository 理解为我们开发时常见到的 DAO，仅仅只是换个叫法罢了），继承自 Spring Data Solr 中的 SolrCrudRepository 接口，示例代码如下：

```
public interface ProductRepository extends SolrCrudRepository<Product,String> {}
```

以上的两个泛型依次表示 POJO 类型和 UniqueKey 域对应的 Java 类型，ProductRepository 接口中几乎不需要定义额外的接口方法，因为 SolrCrudRepository 接口中已经预先定义了索引操作相关的 CRUD 接口方法。然后编写 ProductRepository 的实现类，示例代码如下：

```
@Repository("productRepository")
public class ProductRepositoryImpl extends SimpleSolrRepository<Product,Stri
```

```

ng> implements ProductRepository {
    @Resource
    private SolrTemplate solrTemplate;
    @Autowired(required = true)
    public ProductRepositoryImpl(@Qualifier(value="solrTemplate") SolrOperations
solrOperations) {
        super(solrOperations);
    }
}

```

跟平常的 Web 开发流程类似，我们紧接着需要编写一个 Service 接口：

```

public interface ProductService {
    public void addProductToSolr(Product product);
    public void deleteProductById(String id);
}

```

然后编写 ProductService 的实现类，示例代码如下所示：

```

@Service
public class ProductServiceImpl implements ProductService {
    @Autowired
    private ProductRepository productRepository;
    @Transactional
    public void addProductToSolr(Product product) {
        productRepository.save(product);
    }
    @Transactional
    public void deleteProductById(String id) {
        productRepository.delete(id);
    }
}

```

@Service 和 @Autowired 注解我想大家都不陌生，在 Service 实现类中通过 Spring IOC 注入了 ProductRepository 接口，就好比在 Web 开发中 Service 实现类需要注入 DAO 是一样的。同理，还可以实现 Update 操作，不过由于 Solr 中没有更新索引操作，实际索引更新其实是删除然后再重新添加，示例代码如下所示：

```

@Transactional
public void updateProductToSolr(Product product) {
    Product oldProduct = productRepository.findOne(product.getId());
    oldProduct.setProductName(product.getProductName());
    productRepository.save(oldProduct);
}

```

然后我们可以编写一个 Junit 测试类对 Service 接口进行单元测试，示例代码如下所示：

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {HttpSolrContext.class})
// @ContextConfiguration(locations = "classpath:applicationContext-solr.xml")
public class ProductServiceTest {

```

```

@Autowired
private ProductService productService;
@Test
public void addProductToSolr() {
    Product product = new Product("2", "Product-02");
    productService.addProductToSolr(product);
}
}

```

然后我们继续在 `ProductService` 接口中追加几个接口方法，如下所示：

```

public Product findProductById(String id);
public SolrResultPage<Product> findProductsByIds(List<String> ids);
public SolrResultPage<Product> findAllProducts();

```

以上 3 个接口方法依次表示根据 ID 查询、根据多个 ID 批量查询以及查询所有 `Product` 信息。然后我们需要在 `ProductServiceImpl` 中对其进行实现，示例代码如下：

```

public Product findProductById(String id) {
    return productRepository.findOne(id);
}
public SolrResultPage<Product> findProductsByIds(List<String> ids) {
    return (SolrResultPage<Product>) productRepository.findAll(ids);
}
public SolrResultPage<Product> findAllProducts() {
    return (SolrResultPage<Product>) productRepository.findAll();
}

```

其实 `Spring Data` 中的 `Repository` 接口完全可以没有实现类，`Spring Data` 会自动根据接口的方法名和方法的参数或者根据用户在接口方法上定义的注解来自动构造 `Query`。请看以下这个示例：

```

/**
 * Created by Lanxiaowei
 * ProductRepository 接口，不带实现类
 */
public interface ProductRepositoryWithNoImpl extends SolrCrudRepository<Product, String> {
    Page<Product> findByProductName(String productName, Pageable pageable);
}

```

上面我们定义了 `findByProductName` 方法，其中 `findBy` 是固定前缀，后面是你想要查询的域，因为是面向对象编程，所以这里实际表示的是 `POJO` 的属性，后面的 `Pageable` 参数用于分页和排序。就这样，一个分页接口方法就编写完毕了，不需要为添加 `ProductRepositoryWithNoImpl` 实现类。当然你还可以根据两个域来查询，比如：

```

public interface PersonRepository extends Repository<User, Long> {
    List<Person> findByFirstnameAndLastname(EmailAddress firstname, String lastname);
}

```


以上的示例表示根据 `firstname` 和 `lastname` 进行 AND 查询，Spring Data 会根据 `findBy` 和 AND 自动分割方法名来判断需要查询的域，类似 `findBy` 的方法名固定前缀还有 `findBy`, `find`, `readBy`, `read`, `getBy`, `get` 等，类似 And 的固定域名称连接字符串还有 `Between`, `LessThan`, `GreaterThan`, `Like`，依次表示区间查询、小于、大于、前缀模糊查询。

13.12.3 Spring Data Solr 中的查询

Spring Data Solr 中的 Repository 接口方法有 2 种定义方式，第一种是通过内置的方法名称解析规则自动生成 Query，表 13-3 详细列举了 Spring Data Solr 中支持的方法名称关键字以及它们底层所表示的 Solr 查询表达式。

表 13-3 Spring Data Solr 支持的方法名称关键字及其 Solr 查询表达式

关键字	示 例	Solr 查询表达式
And	<code>findByNameAndPopularity</code>	<code>q=name:?1 AND popularity:?2</code>
Or	<code>findByNameOrPopularity</code>	<code>q=name:?1 OR popularity:?2</code>
Is	<code>findByName</code>	<code>q=name:?1</code>
Not	<code>findByNameNot</code>	<code>q=-name:?1</code>
Between	<code>findByPopularityBetween</code>	<code>q=popularity:[?1 TO ?2]</code>
LessThan	<code>findByPopularityLessThan</code>	<code>q=popularity:[* TO ?1}</code>
LessThanEqual	<code>findByPopularityLessThanEqual</code>	<code>q=popularity:[* TO ?1]</code>
GreaterThan	<code>findByPopularityGreaterThan</code>	<code>q=popularity:{?1 TO *]</code>
GreaterThanEqual	<code>findByPopularityGreaterThanEqual</code>	<code>q=popularity:[?1 TO *]</code>
Before	<code>findByLastModifiedBefore</code>	<code>q=last_modified:[* TO ?1}</code>
After	<code>findByLastModifiedAfter</code>	<code>q=last_modified:{?1 TO *]</code>
Like	<code>findByNameLike</code>	<code>q=name:?1*</code>
NotLike	<code>findByNameNotLike</code>	<code>q=-name:?1*</code>
StartingWith	<code>findByNameStartingWith</code>	<code>q=name:?1*</code>
EndingWith	<code>findByNameEndingWith</code>	<code>q=name:*?1</code>
Containing	<code>findByNameContaining</code>	<code>q=name:*?1*</code>
Matches	<code>findByNameMatches</code>	<code>q=name:?1</code>
In	<code>findByNameIn(Collection<String> names)</code>	<code>q=name:(?1...)</code>
NotIn	<code>findByNameNotIn(Collection<String> names)</code>	<code>q=-name:(?1...)</code>
Within	<code>findByStoreWithin(GeoLocation, Distance)</code>	<code>q={!geofilt pt=?1.latitude,?1.longitude sfield=store d=?2}</code>
Near	<code>findByStoreNear(GeoLocation, Distance)</code>	<code>q={!bbox pt=?1.latitude,?1.longitude sfield=store d=?2}</code>
Near	<code>findByStoreNear(BoundingBox)</code>	<code>q=store[?1.start.latitude,?1.start.longitude TO ?1.end.latitude,?1.end.longitude]</code>
True	<code>findByAvailableTrue</code>	<code>q=inStock:true</code>
False	<code>findByAvailableFalse</code>	<code>q=inStock:false</code>
OrderBy	<code>findByAvailableTrueOrderByNameDesc</code>	<code>q=inStock:true&sort=name desc</code>

也就是说，只要你的 Repository 接口方法名称遵循上面的隐式命名规则，那么 Spring Data Solr 会自动帮你生成相应的 Solr 查询表达式，而完全不用去实现 Repository 接口，节省了很多编码量。

除了可以使用 Spring Data Solr 提供的这种根据方法名称来自动生成 Solr Query 的策略之外，还可以使用 @Query 注解来手动指定 Solr 查询表达式，示例如下：

```
public interface ProductRepository extends SolrRepository<Product, String> {
    // 这里的 ?0 表示参数占位符，?0 即表示引用接口方法中的第零个参数值，同理还有 ?1, ?2
    @Query("inStock:?0")
    List<Product> findByAvailable(Boolean available);
}
```

@Query 注解除了支持普通的 Query 查询，还支持 Filter Query，示例如下：

```
@Query(value="tags:(?0) AND (?1)", filters = {"price:[0 TO 3000000]"},
"images:[* TO *]"))
Page<SolrProduct> findByKeywords ( String keywords, Pageable pageable);
```

@Query 注解的 value 属性表示 Solr 查询中的 q 参数，filters 属性表示 Solr 查询中的 fq 参数，同理还有 fields 属性表示 fl 参数，比如 fields = {"name", "id"} 即表示当前接口查询返回 name 和 id 两个域。defaultOperator 属性表示 q.op 参数，defType 属性表示 defType 参数，requestHandler 属性表示当前接口方法将由 Solr Server 的哪个 Request Handler 来处理，比如 requestHandler="/select" 表示查询，requestHandler="/update" 表示索引添加更新删除。timeAllowed 属性用来表示当前接口方法的最大执行超时时间。

除了可以采用将 Solr 查询表达式通过 @Query 注解显式的定义在对应的注解属性上之外，还可以将 Solr 查询表达式定义在外部的 properties 属性文件中，然后根据 name 值进行引入，示例如下：

```
#solr-named-queries.properties 属性文件
Product.findByNameQuery=popularity:?0
Product.findByName=name:?0
```

Repository 接口方法可以这样定义：

```
public interface ProductRepository extends SolrCrudRepository<Product, String> {
    List<Product> findByNameQuery(Integer popularity);
    @Query(name = "Product.findByName")
    List<Product> findByNameQuery(String name);
}
```

findByNameQuery 方法虽然没有通过 @Query 注解的 name 属性值显式的来引用外部 properties 属性文件中的 Solr 查询表达式，但是 Spring Data Solr 会自动根据 properties 属性文件中定义的前缀 Product 找到 ProductRepository 接口，然后找到 findByNameQuery 方法。也可以通过 @Query 注解的 name 属性显式的指定 Solr 查询表达式的 key。此时需要在

src/main/resources 下新建 META-INF 目录，然后在新建的 META-INF 目录下创建一个 solr-named-queries.properties 属性文件，属性文件中配置内容即命名查询 key 和 Solr 查询表达式 value。Repository 接口方法上的 @Query 注解通过 name 属性值来引用这里的 key。如果你没有通过 @Query 注解的 name 属性值指定 key，那么默认 Spring Data Solr 会根据你当前 Repository 接口对应的 POJO 类名称和当前接口方法名称拼接组成一个 key 即“POJO 类名.接口方法名”。当然 solr-named-queries.properties 属性文件中的 key 你可以随意定义，此时你就需要显式的通过 @Query 注解的 name 属性值指定 key 来引用 Solr 查询表达式啦！

如果你需要分页查询，那么你可以传入 Pageable 参数，下面是在查询方法中使用 Pageable 和 Sort 参数的定义示例：

```
Page<User> findByLastname(String lastname, Pageable pageable); // 分页且排序
List<User> findByLastname(String lastname, Sort sort);           // 不分页只排序
List<User> findByLastname(String lastname, Pageable pageable); // 分页且排序
```

如果想要实现 Facet 查询，那么你还可以为接口方法添加 @Facet 注解，示例如下：

```
@Query(value = "*:~")
@Facet(fields = { "name" }, limit = 5)
FacetPage<Product> findAllFacetOnPopularity(Pageable page);
```

@Facet 注解支持的属性有：fields 即你要对哪些域进行 Facet 查询，limit 属性对应 facet.limit 参数，queries 属性对应 facet.query 参数，minCount 属性对应 facet.minCount 参数，limit 属性对应 facet.limit 参数，prefix 属性对应 facet.prefix 参数，pivots 属性对应 facet.pivot 参数，pivotMinCount 对应 pivot.mincount 参数，pivotMinCount 对应 pivot.mincount 参数，不过 @Facet 注解支持的 Facet 参数还不够全面，如果你需要设置的参数 @Facet 注解不支持，那么此时需要自定义 Repository 接口，然后注入 SolrTemplate 工具类自己实现。关于如何自定义 Repository 接口下一章再做详细讲解。

此外你还可以为你的 Repository 接口方法添加 @Highlight 注解，从而开启 Solr 的高亮功能，不过此时你的接口方法返回值需要修改为 HighlightPage 类型，这样才能通过 HighlightPage 类获取到 Solr Server 返回的高亮信息，下面是高亮查询的一个简单示例：

```
@Highlight(prefix = "<b>", postfix = "</b>")
@Query(value="popularity:[?1 TO ?0]")
HighlightPage<Product> findByPopularityBetween(int max,int min,Pageable page);
```

13.12.4 Spring Data Solr 中的 Repository 详解

Spring Data Solr 中的 Repository 其实就是数据访问层的一个抽象，我们通常称为 DAO 层，不过在 Spring Data 家族中叫做 Repository。Repository 的接口设计是以领域驱动模式来设计，完全以面向对象的方式来操作 Solr。其中将 Solr 中的 Document 映射为 POJO，然后每个 Repository 接口都要通过泛型参数来指定当前 Repository 操作目标是哪个 POJO，Spring Data

Solr 会自动根据你在 POJO 中添加的 `@SolrDocument`、`@Field` 注解判断出实际需要操作的 Solr Core 以及域。为了简化开发，Spring Data Solr 底层帮用户将 Solr 中的一些常见操作进行封装，比如添加更新删除查询索引文档、分页、排序等，因此一般建议用户自定义的 Repository 接口直接继承 `SolrCrudRepository` 接口，无须自己重新进行底层操作封装。Spring Data Solr 默认提供了如图 13-6 所示的几种 Repository 实现，其中 `PagingAndSortingRepository` 用于分页排序，需要分页和排序功能时你需要继承此接口。

如果你认为 Spring Data Solr 的 Repository 接口封装的不够好，可以直接继承 Repository 接口，自己重新封装。Spring Data Solr 中的 Repository 接口是个象征性的空接口，里面没有定义任何接口方法，因此你可以自定义接口方法，示例如下：

```
public interface MyCrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {
    T save(T entity);
    T findOne(ID primaryKey);
    Iterable<T> findAll();
    Long count();
    void delete(T entity);
    boolean exists(ID primaryKey);
    // ..... 省略
}
```

有了 Spring IOC 环境的支持，我们的 Repository 可以通过在 Spring 配置文件中通过包扫描的方式，交给 Spring IOC 自动实例化并管理，这样也便于后续 Service 中注入需要依赖的 Repository 接口，配置示例如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:solr="http://www.springframework.org/schema/data/solr"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/data/solr
        http://www.springframework.org/schema/data/solr/spring-solr-1.0.xsd">
    <solr:repositories base-package="com.xxx.repositories" />
</beans>
```

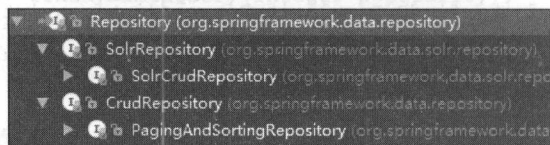


图 13-6 Spring Data Solr 中内置的各种 Repository 实现

上面的 `base-package="com.xxx.repositories"` 表示 Spring Data Solr 会在 `base-package` 属性配置的包路径下扫描寻找所有 Repository 接口，然后你就可以其他地方通过 IOC 注解注

入你需要的 Repository 即可，比如：

```
public class XXXXServiceImpl {
    @Autowired
    private PersonRepository repository;
```

自 Spring4 开始，支持 Java Config 方法来代替传统的 Spring XML 方式来配置 Spring，你只需要新建一个普通类添加一个 @Configuration 注解，即表明当前类是一个 Spring IOC 容器，然后你可以使用 @EnableSolrRepositories 注解来实现 Spring Data Solr 中的 Repository 扫描，等价于 Spring XML 中的 <repositories base-package.../> 配置。如果你热衷于这种配置方式，那么可以考虑下这种方式。示例代码如下：

```
@Configuration
@EnableSolrRepositories(basePackages = {
    "com.yida.solr.book.examples.ch13.springdatasolr.repository"},
    multicoreSupport = true)
class ApplicationConfiguration {
    @Bean
    public EntityManagerFactory entityManagerFactory() {
        // ...
    }
}
```

将 Repository 交给 Spring IOC 管理之后，你还可以以编码的方式通过 SolrRepository Factory 工厂类获取到已经在 Spring IOC 中注册的 Repository 实例，示例代码如下：

```
RepositoryFactorySupport factory = new SolrRepositoryFactory(solrClient);
UserRepository repository = factory.getRepository(UserRepository.class);
```

默认情况下，Spring Data Solr 是推崇用户以无 Repository 接口实现的方式来编程，利用底层自动生成 Query 或者为 Repository 接口方法添加 @Query 注解方式来作为隐式实现。如果你觉得默认的这种无 Repository 接口实现的方式约束了你，你可以自定义 Repository 接口实现类，如果你的 Repository 实现类类名称有固定的后缀，那么你可以通过 repository-impl-postfix（限定 Repository 实现类的后缀）属性进一步限定包扫描范围，配置示例如下：

```
// repository-impl-postfix 属性的默认值就是 Impl
<repositories base-package="com.acme.repository" repository-impl-postfix="Impl" />
```

除了开启包扫描方式自动注册 Repository，你可以显式的为 Repository 接口实现类添加 @Repository 注解，还可以在 Spring XML 中通过 <bean id="xxxx".../> 这种最原始的方式进行 Bean 注册。

当采用自定义 Repository 接口实现类，意味着需要注入 Spring Data Solr 提供的 SolrTemplate 工具类自己封装一些常用操作，为了便于获取 SolrTemplate 实例，显然此时还是离不开 Spring IOC，需要在自定义的 Repository 接口实现类中注入 SolrTemplate 对象，首先需要在 Spring IOC 容器中注册 SolrTemplate，下面是 SolrTemplate 注册配置示例：

```

<!-- EmbeddedSolrServer 配置 -->
<solr:embedded-solr-server id="solrServer" solrHome="classpath:com/acme/solr" />
<solr:solr-client id="solrClient" url="${solr.server.url}" />
<bean id="solrTemplate" class="org.springframework.data.solr.core.SolrTemplate">
<constructor-arg index="0" ref="solrClient"/>
</bean>

```

这里我们在 Spring 配置文件中将 SolrTemplate 实例对象注册到 Spring IOC 容器，bean 的 ID 为 solrTemplate，你就可以在任意的 Repository 接口实现类中通过 @Autowired 注解引用 SolrTemplate 实例：

```

@Autowired
private SolrTemplate solrTemplate;

```

这里的 SolrTemplate 类似于 Spring Data JDBC 里的 JdbcTemplate，它是 Solr 相关操作的工具类。关于 SolrTemplate 工具类如何使用，请继续学习下一小节。

13.12.5 Spring Data Solr 中 SolrTemplate 工具类详解

SolrTemplate 是 Spring Data Solr 为简化 Solr 客户端操作，基于 SolrJ API 又进行了一层薄封装，内部提供的工具方法与 SolrClient 基本类似。对于简单的查询，通过前面章节的学习，我们已经知道，可以通过定义接口方法名称能够自动生成 Query 或者通过 @Query 注解方式明确指定查询表达式，但是这两个方式都有不足之处。第一种方式对于复杂查询无能为力，第二种方式虽然能够解决第一种方式的不足之处，注解支持的功能毕竟有限，比如 join 查询就无法通过注解来实现，而且也不是所有的查询参数都能通过注解以及注解属性来设置。此时，你就需要借助 SolrTemplate 工具类以编程的方式来手动实现复杂的 Solr 请求。下面将逐一详细介绍如何使用 SolrTemplate 来完成 Solr 中的各种请求操作。

1. SolrTemplate 实例化

你除了可以直接在 Spring 配置文件中注册 SolrTemplate，然后交由 Spring IOC 容器帮助我们管理 SolrTemplate 实例，也可以直接通过 SolrTemplate 的构造函数手动构造 SolrTemplate 实例对象。

创建 SolrTemplate 实例，首先必须提供一个 SolrClient 对象，这也是创建 SolrTemplate 实例的最简单方式，如下所示：

```

SolrTemplate template = new SolrTemplate(new HttpSolrClient("http://localhost:8080/solr"));

```

你也可以通过传入一个 SolrClientFactory 实例来构造 SolrTemplate，SolrClientFactory 接口主要有 3 个子类：EmbeddedSolrServerFactoryBean、HttpSolrClientFactoryBean 和 MulticoreSolrClientFactory。EmbeddedSolrServerFactoryBean 用于创建 EmbeddedSolrServer 实例，HttpSolrClientFactoryBean 用于创建 HttpSolrClient 或者 LBHttpSolrClient 实例，当传入的 url 参数是多个采用逗号分隔的 Solr

Server 访问 URL 时会自动创建 LBHttpSolrClient 实例。MulticoreSolrClientFactory 用于多 Core 情况下 SolrClient 实例的管理，内部会使用一个 Map 来管理每个 Core 对应的 SolrClient 实例对象，key 为 Core 名称，Value 为 SolrClient 实例对象。准确来说，MulticoreSolrClientFactory 其实并不会生成 SolrClient 实例对象，它只是每个 Core 对应的 SolrClient 实例对象的管理者。构建 EmbeddedSolrServerFactoryBean 你只需要传入一个 solrHome 参数即可或者通过 SetSolrHome() 方法设置。在 Spring XML 中，可以通过 <constructor-arg> 或 <property> 元素传入 solrHome 参数。构建 HttpSolrClientFactoryBean 必选的参数只有 url，它表示 Solr Server 的访问 URL。当然你还可以额外设置 timeout、maxConnections、singleton 等参数，其中 timeout 参数表示 Http 连接的最大超时时间，maxConnections 参数表示 Http 最大连接数，singleton 参数表示通过 HttpSolrClientFactoryBean 创建的 SolrClient 实例是否保持单例。SolrTemplate 构造函数中还有一个特殊的 SolrConverter 参数需要引起你的注意。它主要用于 POJO 与 SolrDocumentList 之间的对象类型转换，默认 Spring Data Solr 会根据你在 POJO 上定义的注解信息进行自动转换，当自动类型转换不能满足你的需求时，你可能需要实现 SolrConverter 接口，重写实现你自己的类型转换逻辑。

当你的查询复杂时，必然需要自定义 Repository 实现类，并在 Repository 实现类中注入 SolrTemplate 实例，但是 SolrTemplate 类内部封装的方法都是通过 SolrClient 实现的，然而 SolrTemplate 在调用 SolrClient 的方法时并没有传入 Core 名称，所以当你想要灵活的切换操作的目标 Core 时，就显得无能为力了，不够灵活。因为默认 Spring Data Solr 会自动根据 POJO 类上的 @SolrDocument 注解中 coreName 属性获取 core 名称，然后自动在 Solr Server 访问 URL 后面加上 Core 名称。由于我们现在是通过自定义 Repository 实现类，无法利用这一特性，因此我们需要自己去解析 POJO 类上的 @SolrDocument 注解，获取 Core 名称，然后重写 SolrTemplate 类，重点是重写其 setSolrCore 方法，在 setSolrCore 方法内部通过 solrClient 实例获取到 Solr Server 访问的 BaseUrl，然后将用户设置的 Core 名称追加到 BaseUrl 后面或者替换 URL 中已有的 Core 名称，当用户通过 setSolrCore 方法将 Core 名称置空，并将 Core 名称从 URL 上移除，这样就能实现在查询时，操作的目标 Core 能随时切换。我自定义实现的 CustomSolrTemplate 类的部分源码（完整源码请从随书源码中获取），如下所示：

```
/**
 * Created by Lanxiaowei
 * 重写默认的 SolrTemplate, 使其能够灵活的设置操作的目标 Core
 */
public class CustomSolrTemplate extends SolrTemplate {
    ...// 省略
    /**
     * 将 Core 名称追加到 Solr Server 的访问 BaseURL 上
     * @param core
     * @param solrClient
     */
}
```

```

private void appendCoreToBaseUrl(String core, SolrClient solrClient) {
    if(StringUtils.isEmpty(core) && this.isHttpSolrClient(solrClient)) {
        HttpSolrClient httpSolrClient = (HttpSolrClient)solrClient;
        // 如果当前 BaseUrl 上已经追加了其他 Core 名称, 则需要将其替换为新设置的 core 名称
        if(StringUtils.isEmpty(this.solrCore)) {
            resetCoreFromBaseUrl(this.solrCore, httpSolrClient);
        }
        String url = SolrClientUtils.appendCoreToBaseUrl(httpSolrClient.getBase
URL(), core);
        httpSolrClient.setBaseUrl(url);
    }
}
/**
 * 将 Core 名称从 Solr Server 访问 URL 上剔除掉
 * @param core
 * @param solrClient
 */
private void resetCoreFromBaseUrl(String core, SolrClient solrClient) {
    if(StringUtils.isEmpty(core) && this.isHttpSolrClient(solrClient)) {
        HttpSolrClient httpSolrClient = (HttpSolrClient)solrClient;
        String baseUrl = httpSolrClient.getBaseUrl();
        // 如果是 BaseUrl 即还没有在后面追加 Core 名称
        if(isBaseUrl(baseUrl)) {
            return;
        }
        if(null == core || "".equals(core)) {
            if(null != this.solrCore && !"".equals(this.solrCore)) {
                baseUrl = getBaseUrl(baseUrl);
            }
        } else {
            baseUrl = baseUrl.substring(0, baseUrl.indexOf(core) - 1);
        }
        httpSolrClient.setBaseUrl(baseUrl);
    }
}
@Override
public void setSolrCore(String solrCore) {
    if(null == solrCore || "".equals(solrCore)) {
        // 若初始化 SolrTemplate 时未指定 Core
        if(null == this.solrCore || "".equals(this.solrCore)) {
            return;
        }
        resetCoreFromBaseUrl(solrCore, this.getSolrClientFactory().getSolrClient());
        this.solrCore = solrCore;
        return;
    }
    appendCoreToBaseUrl(solrCore, this.getSolrClientFactory().getSolrClient());
    this.solrCore = solrCore;
}

```

此时重点就落在如何获取当前 Repository 实现类默认操作的 Core 上, 我们知道, 每

个 Repository 接口声明时都会添加 POJO 泛型参数, 可以通过反射获取到 POJO 类型, 然后通过 POJO 的 class 类型的 getAnnotation() 方法可以获取到 POJO 类上面定义的 @SolrDocument 注解, 最后调用注解的 solrCoreName() 方法就能获取到 Core 名称, 前提是用户已经为 @SolrDocument 注解添加了 solrCoreName 属性。如果用户未指定 solrCoreName 属性, 那么此时你可以以 POJO 类的名称为默认操作 Core, 但前提是你必须提前约定好: Core 名称必须 POJO 类名称保持一致。具体实现的示例代码如下所示:

```
/**
 * Created by Lanxiaowei
 * 自定义的 Repository 接口, 其他 Repository 接口需要继承此接口
 */
```

```
public interface BaseRepository <T, ID extends Serializable> {
    public void addDoc(T t);
    public void addDoc(T t,String core);
    public void updateDoc(T t);
    public void updateDoc(T t,String core);
}
```

BaseRepository 实现类的部分源码如下所示:

```
/**
 * Created by Lanxiaowei
 * 自定义的 Repository 接口实现类, 其他 Repository 接口实现类需要继承此基类
 */
```

```
public class BaseRepositoryImpl<T, ID extends Serializable> implements BaseRepository<T, ID> {
    @Autowired
    private CustomSolrTemplate solrTemplate;
    /** 默认操作的目标 Core 名称 */
    private String coreName;
    /** 当前 Repository 对应的 POJO 类型 */
    private Class entityClass;
    /** 获取 POJO 的主键域名称 */
    private String uniqueKey;
    public BaseRepositoryImpl(){
        this.entityClass = (Class<T>)((ParameterizedType)this.getClass().getGenericSuperclass()).getActualTypeArguments()[0];
        SolrDocument solrDocument = (SolrDocument)this.entityClass.getAnnotation(
            SolrDocument.class);
        if(null != solrDocument) {
            String core = solrDocument.solrCoreName();
            if(null == core || "".equals(core)) {
                // 如果用户未为 @SolrDocument 注解设置 solrCoreName 属性
                this.coreName = entityClass.getSimpleName();
            } else {
                this.coreName = core;
            }
        }
        Field[] fields = this.entityClass.getDeclaredFields();
        for(Field f : fields) {
```

```

        f.setAccessible(true);
        // 查找被 @Id 注解修饰的属性
        if (f.isAnnotationPresent(Id.class)) {
            this.uniqueKey = f.getName();
            break;
        }
    }
}

```

然后你就可以自己根据 `solrTemplate` 工具类封装底层的 Solr 操作，因为我们已经实现了通过调用自定义的 `CustomSolrTemplate` 类的 `setSolrCore` 方法来自由切换 Core，因此多 Core 操作场景下，编码也变得更加灵活了。

2. 使用 SolrTemplate 实现索引的 CRUD 操作

上一小节中，我们已经自定义了 `BaseRepositoryImpl` 基类，并注入了自定义的 `CustomSolrTemplate` 工具类。通过 `CustomSolrTemplate` 工具类提供的很多工具方法，常见的 CRUD 操作变得十分简单，比如添加一个索引文档，可以调用 `CustomSolrTemplate` 的 `saveDocument` 方法，同理还有 `saveDocuments` 方法用于批量添加索引文档，还可以为 `saveDocument` 方法指定 `commitWithinMs` 参数，即启用软提交模式，这些与原生的 SolrJ 操作几乎如出一辙。当然还可以以面向对象的方式直接通过调用 `addBean` 或 `addBeans` 来添加一个 POJO。至于索引文档更新，其实就是根据 `UniqueKey` 自动判断添加还是“更新”。删除索引文档有 `deleteById` 和 `delete(SolrDataQuery query)` 这跟 SolrJ 也很相似，只不过此时的人参不再是 SolrJ 中的 `SolrQuery`，而是 Spring Data Solr 中的 `SolrDataQuery` 类型。`SolrDataQuery` 是 Spring Data Solr 中的查询抽象，而查询必须要有查询条件，Spring Data Solr 中的查询条件使用 `Criteria` 类进行表示。比如查询 `title` 域上包含 "solr" 的索引文档，用 `Criteria` 类表示就是：`newCriteria("title").contains("solr")`。除了 `contains` 方法之外，`Criteria` 还提供了 `startsWith`（前缀查询）、`endsWith`（后缀查询）、`between`（区间范围）、`lessThan`（小于）、`fuzzy`（对应 fuzzy query）、`near`（对应地理空间查询）、`function`（对应 Function Query）等方法，同时还有 `and`、`or` 函数用于连接多个 `Criteria`。然后你就可以将 `Criteria` 对象传入 `SimpleQuery` 类的构造函数，创建一个 `SimpleQuery` 对象（`SimpleQuery` 是 `SolrDataQuery` 的一个子类），示例代码如下所示：

```
SimpleQuery search = new SimpleQuery(criteria);
```

`SolrDataQuery` 拥有如图 13-7 所示的几种实现类，大致分 5 类：`FilterQuery`、`TermsQuery`、`SimpleQuery`（普通的简单查询）、`FacetQuery`、`HighlightQuery`。

下面是一个 `FacetQuery` 的简单使用示例，代码如下所示：



图 13-7 SolrDataQuery 的实现类

```
FacetQuery query = new SimpleFacetQuery(new Criteria(Criteria.WILDCARD).
expression(Criteria.WILDCARD))
    .setFacetOptions(new FacetOptions().addFacetOnField("name").setFacetLimit(5));
FacetPage<Product> page = solrTemplate.queryForFacetPage(query, Product.
class);
```

以上的实例中 `new Criteria(Criteria.WILDCARD).expression(Criteria.WILDCARD)` 部分等价于设置 `q` 参数等于 `*:*`，即查询所有索引文档，然后设置 `facet.field=name` 以及 `facet.limit=5`。

下面是一个 `FilterQuery` 的简单使用示例，代码如下所示：

```
Query query = new SimpleQuery(new Criteria("category").is("xx"));
FilterQuery fq = new SimpleFilterQuery(new Criteria("store")
    .near(new GeoLocation(48.305478, 14.286699), new Distance(5)));
query.addFilterQuery(fq);
```

上面的查询示例中，`new Criteria("category").is("supercalifragilisticexpialidocious")` 部分表示设置 `q` 参数等于 `category:xx`，然后在 `store` 域上执行地理空间查询，即离 (48.305 478, 14.286 699) 坐标点方圆 5 千米以内的索引文档。

下面是一个 `HighlightQuery` 的简单使用示例，代码如下所示：

```
SimpleHighlightQuery query = new SimpleHighlightQuery();
// 相当于设置 fl 参数
query.addProjectionOnFields("id title");
Criteria conditions = new Criteria("title").contains(term)
    .or("description").contains(term)
    .or("content").contains(term);
query.addCriteria(conditions);
HighlightOptions hlOptions = new HighlightOptions();
// 相当于设置 hl.fl 参数
hlOptions.addField("content", "title", "description");
query.setHighlightOptions(hlOptions);
HighlightPage<SolrManagedWebResource> test = solrTemplate.queryForHighlightPage(query,
Product.class);
```

如果你想要在 Spring Data Solr 中实现原子更新，那么此时你需要使用 `PartialUpdate` 类来构造原子更新请求，示例代码如下：

```
PartialUpdate update = new PartialUpdate("id", "123");
update.add("name", "your-new-name");
solrTemplate.saveBean(update);
```

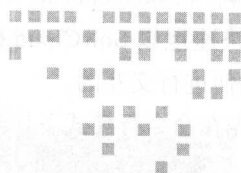
3. 使用 SolrTemplate 执行任意 Solr 请求

当 Spring Data Solr 提供的 API 无法表示你所要执行的 Solr 查询，或者你想要执行的 Solr 查询表达式过于复杂而你对 Spring Data Solr API 不够熟悉，那么此时你需要使用 `SolrCallback<T>` 接口。`SolrCallback` 接口只有一个接口方法 `doInSolr(SolrClient solrClient)`，入参是 `SolrClient` 实例对象，有了 `SolrClient` 对象，就可以以纯粹 SolrJ 的方式来与 Solr Server 进行交互了，从而彻底

摆脱 Spring Data Solr 框架的束缚，当你对 Spring Data Solr 不熟悉，或者对这种为了面向对象而面向对象的编程方式已经厌烦，此时你可以实现 `SolrCallback<T>` 接口并重写 `doInSolr` 方法，这里的泛型 `T` 即返回值类型。实现 `SolrCallback<T>` 接口你可以新建一个类，不过一般推荐采用匿名实现类的方式比较方便。`SolrTemplate` 工具类提供了一个 `execute(SolrCallback<T> action)` 方法，它接收一个 `SolrCallback` 接口实现类，然后将 `SolrClient` 实例对象传入 `doInSolr` 方法，然后执行 `doInSolr` 方法返回响应结果。使用示例如下所示：

```
this.solrTemplate.execute(new SolrCallback<QueryResponse>() {
    public QueryResponse doInSolr(SolrClient solrClient) throws SolrServerException,
    IOException {
        SolrQuery query = new org.apache.solr.client.solrj.SolrQuery();
        query.setRequestHandler("/select");
        query.set("q", "type:book");
        query.set("fl", "id,brand,color,size,score");
        query.set("indent", "true");
        QueryResponse response = solrClient.query(query, SolrRequest.METHOD.GET);
        return response;
    }
});
```

采用这种方式就好比在 Hibernate 里执行原生的 SQL 语句，通过这种方式，你可以执行任意的 Solr 查询，而不用关心 Spring Data Solr API 如何使用。



第 14 章

Chapter 14

SolrCloud

通过第 14 章，你将可以学习到以下内容：

- ❑ SolrCloud 快速入门；
- ❑ Core 与 Collection 的区别；
- ❑ 使用 Zookeeper 管理 SolrCloud 配置文件；
- ❑ SolrCloud 的分布式索引和查询；
- ❑ 使用 Solr Collection API；
- ❑ Solr 索引主从复制；
- ❑ 跨数据中心的索引复制 (CDCR)。

在本章中，我们将学习如何使用 SolrCloud 来设计、配置以及操作大规模的 Solr 集群。本章可能会有一些挑战性，因为会有一些新的概念和术语你并不熟悉。但是我确信，当你看完本章之后，你一定会对 Solr 集群有一个深刻的全方位理解，并且能够自行搭建一个健壮的大规模 Solr 分布式集群。本章的理论多于实践，请读者多注重理解，适当上机实践可以加深你的理解。

14.1 SolrCloud 快速入门

SolrCloud 是设计用来处理跨多台服务器的分布式索引和查询工具，具有高可用性、可扩展性、自动容错性的特点。在 SolrCloud 中，索引数据被分成多个 Shard (分片)，而每个 Shard 可以托管在多台机器上，同时为每个 Shard 提供副本冗余来提供可扩展性和自动容错

性。SolrCloud 利用 Zookeeper 来管理集群中所有节点以及集中管理集群配置文件。本节我们将学习如何快速地以 SolrCloud 模式启动 Solr，这样你就能直观感受到在索引和查询时 Shard 之间是如何进行交互的。

首先我们先在命令行模式下切到 Solr 安装包解压后的 bin 目录下，然后执行如下命令：

```
solr -e cloud
```

然后，你就会看到如下提示信息：

```
Welcome to the SolrCloud example!
```

下面的提示信息询问我们需要在集群中运行几个节点，默认支持 1 ~ 4 个节点，建议至少设置 2 个，这里我们选择 4 个。这里所谓的多个节点不过是同一个 IP 设置为不同的端口号罢了。然后它会询问我们节点 1 的端口号设置为多少，默认为 8983（即 Jetty 的默认端口号），我们直接回车保持默认即可，接着会询问你第 2 个节点的端口号设置为多少，中括号里是随机生成的端口号，如果你不设置，那么就会采用随机生成的端口号。为了保持端口号的连续性，这里我们设置为 8984。同理，我们设置第 3 个节点的端口号为 8985，第 4 个节点的端口号为 8986。具体操作如图 14-1 所示。

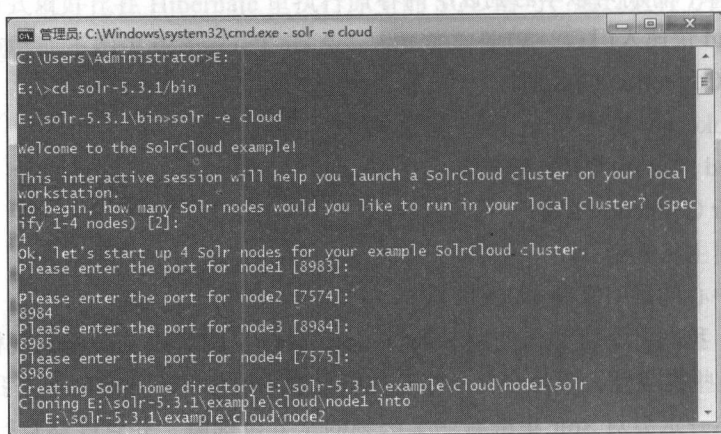


图 14-1 SolrCloud 启动示例图

等待 4 个节点都启动完成之后，它会提示你创建一个新的 Collection，默认 Collection 名称为 gettingstarted，这里我们保持默认即可。然后它会询问你希望将 gettingstarted 这个 Collection 分成几个 Shard，默认为 2，这里我们设置为 4。紧接着它继续询问你每个 Shard 需要设置几个 Replica（分片副本），默认为 2，这里我们设置为 2 个。此时它会询问你选择哪个类型的配置文件，这里我们选择 sample_techproducts_configs，具体操作如图 14-2 所示。最后你将看到如下提示信息：

```
SolrCloud example running, please visit: http://localhost:8983/solr
```

由图 14-2 可知, SolrCloud 集群已经搭建成功, 此时你可以打开浏览器访问 <http://localhost:8983/solr>, Solr 后台管理界面左侧会有个 “Cloud” 菜单, 单击 “Graph”, 然后会出现如图 14-3 所示的画面。

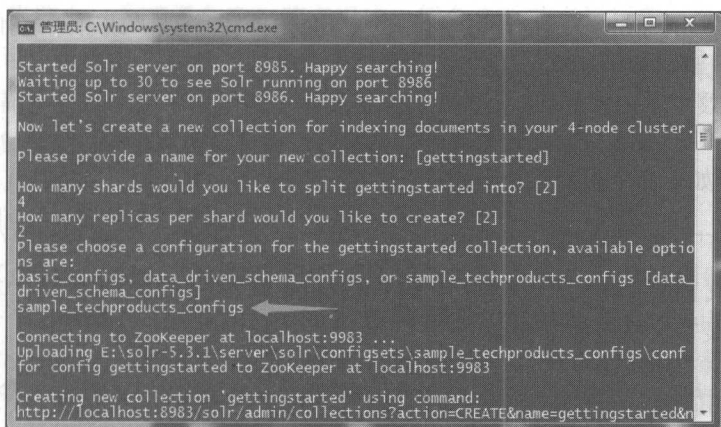


图 14-2 为 Collection 选择配置文件示例图

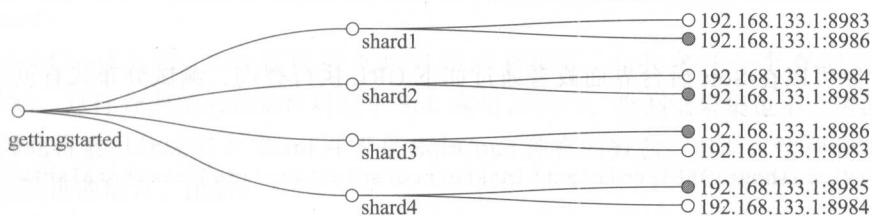


图 14-3 SolrCloud 集群结构示意图

图 14-3 中, gettingstarted 即我们的 Collection, 它是索引文档的集合, 然后我们将 Collection 分割成 4 份, 即 shard1、shard2、shard3、shard4, 每一份叫做一个 Shard (分片), 而每个 Shard 又复制一份形成两份相同的 Shard, 复制而来的新 Shard 我们称为 Replica (副本), 而被复制的 Shard 习惯称为 Master Shard (主分片), 两者分布在两个不同节点上形成 Leader-Follower 结构。

我们可以在命令行下通过 `solr status` 命令查看到集群中每个节点的运行状态, 返回的状态信息如下所示:

```
{
  "solr_home": "E:\\solr-5.3.1\\example\\cloud\\node1\\solr\\",
  "version": "5.3.1 1703449 - noble - 2015-09-17 01:48:15",
  "startTime": "2016-10-29T06:40:07.075Z",
  "uptime": "0 days, 0 hours, 31 minutes, 43 seconds",
  "memory": "100.3 MB (%20.5) of 490.7 MB",
  "cloud": {
```

```
"ZooKeeper":"localhost:9983",
"liveNodes":"4",
"collections":"1"}}
```

ZooKeeper 表示当前 Zookeeper 服务节点的 ip 和端口，liveNodes 表示当前集群中活跃节点的总个数，collections 表示当前集群中的 Collection 总个数。或者通过如下命令获取各个分片和副本的详细健康状态：

```
solr healthcheck -c gettngstarted
```

你可以执行如下命令重启某个节点：

```
solr restart -c -p 8983 -z localhost:9983 -s example/cloud/node1/solr
```

接下来，我们可以尝试为集群中的 collection 添加一些索引文档，首先需要切到 `exampledocs` 目录下，执行如下命令：

```
java -Durl=http://localhost:8983/solr/gettngstarted/update -jar post.jar ipod_
video.xml
java -Durl=http://localhost:8983/solr/gettngstarted/update -jar post.jar ipod_
other.xml
java -Durl=http://localhost:8983/solr/gettngstarted/update -jar post.jar mem.
xml
```

然后你可以在 Solr 后台界面或者通过如下 URL 执行查询，测试分布式查询是否正常工作：

```
http://localhost:8983/solr/gettngstarted/select?q=*&distributed=false&wt=json&indent=true
```

最后你可以执行如下命令停止所有节点：

```
solr stop -all
```

14.2 SolrCloud 工作原理

下面将介绍 SolrCloud 各种特性的工作原理，为了能够理解这些特性，首先你需要理解 SolrCloud 中的一些核心概念。

14.2.1 SolrCloud 的核心概念

SolrCloud 集群是由构建在物理概念之上的一些逻辑概念层组成。

1. 逻辑概念

一个 Solr 集群上可以承载多个 Collection 的索引文档。一个 Collection 由多个索引文档组成，而一个 Collection 可以被分割成多个 Shard (分片)，每个分片中包含了这个 Collection

中的部分索引文档。理论上讲，一个 Collection 包含 Shard 的个数决定了 Collection 中能够合理包含的索引文档总个数以及单个查询请求可能的并行度。

2. 物理概念

一个 Solr 集群由一个或多个 Solr 节点组成，每个 Solr 节点运行着一个 Solr Server 实例。每个 Solr 节点可以包含多个 Core，而集群中的每个 Core 其实就是某个逻辑概念上的分片对应的某一个物理副本。每个副本使用同一套配置文件，每个 Shard 的副本个数决定了 Collection 的冗余级别、集群的自动容错度以及高负载情况下的并发查询请求数量。

14.2.2 SolrCloud 中的 Shard

当单个节点上的 Collection 包含的索引文档过大时，你可以通过创建多个 Shard 进行分别存储。一个 Shard 是一个 Collection 的逻辑部分，它包含了 Collection 中的部分索引文档，因此 Collection 中的每个索引文档都直接属于一个 Shard，每个索引文档分配给哪个 Shard 则取决于你的 Shard 分片策略。比如，你有一个 Collection，其中每个索引文档中都有一个“city”域，那么你可以将 city 域值相同的索引文档分配给同一个 Shard。不同的 Collection 可以简单地对每个索引文档的 UniqueKey 进行 Hash 计算从而决定它属于哪个 Shard。

在 SolrCloud 之前，Solr 就已经支持分布式查询了。Solr 允许一个查询请求跨多个 Shard 执行，这样查询请求就是针对整个 Solr 索引而执行，查询结果集也不会遗漏任何索引文档。因此将索引分成多个 Shard 并不是 SolrCloud 独有的概念。然而，这种分布式查询有很多需要改进的地方，比如：

- ❑ 大多数情况下，将索引分割成多个 Shard 需要手动执行。
- ❑ 不支持分布式索引，这意味着你需要显式地将索引文档发送到指定的 Shard，Solr 不会自动识别每个索引文档属于哪个 Shard。
- ❑ 不支持负载均衡和自动故障转移特性，因此，如果你的查询负载很高，那么你需要显式地将查询指向某个特定的 Shard，而如果某个 Shard 挂掉了而你无从得知，那么你的查询就不可用了。

SolrCloud 解决了上面所有问题，SolrCloud 支持自动的分布式索引和查询，同时借助 Zookeeper 提供了自动故障转移和负载均衡功能。此外，每个 Shard 同时还可以拥有多个 Replica 以提供额外的系统健壮性。在 SolrCloud 模式下，没有 Master 和 Slave 的概念，取而代之的是，每个 Shard 由至少一个物理存在的 Replica 组成，其中一个 Replica 会被选举为 Leader。如果一个 Leader 挂掉了，Zookeeper 会自动以其他 Replica 节点中选举新的 Leader，从而保证集群高可用。当你发送一个索引文档到 Solr 集群的任意一个节点请求创建索引时，Solr 集群首先判断索引文档属于哪个 Shard，然后确定该 Shard 的 Leader 所在节点，紧接着索引文档会转发到当前 Leader 节点上进行索引，最后 Leader 节点会将索引文档

更新广播给其他 Replica。

14.2.3 Collection VS Core

前面章节中，我们一直都在使用 Solr Core。概括来讲，一个 Solr Core 是 Solr Server 中被唯一命名、可配置的、受管理的索引集合的总称。一个 Solr Server 实例可以拥有多个 Solr Core。Solr 中的 Core 通常是通过定义不同的 Schema 来分割索引文档的。SolrCloud 中的 Collection 虽然也是索引集合的总称，但它是逻辑概念。Collection 会将索引文档分割成多个 Shard，这些 Shard 会分布在多个 Solr Server 节点之上，每个 Shard 的 Replica 才是真正物理存在的 Core，而 Collection 并没有物理存在于任意一个节点上。在 SolrCloud 模式下，你不需要再关心物理存在的 Core，你的关注点应该是 Shard。

正如单节点的 Solr Server 可以托管多个 Solr Core，SolrCloud 集群也可以托管多个 Collection，如果你需要使用不同的 Schema 来表示不同的 Document，此时你需要使用不同的 Collection。下面就 SolrCloud 中的一些常见术语做简单解释，便于大家能够理解它们的表示含义以及各个概念术语之间的区别与联系：

- ❑ **Collection**：同一类型的索引文档的集合，但是这些索引文档并不实际存储在同一台机器上，它们通常会被分割成多个分片，然后每个分片会创建多个副本，所以实际存储的是副本，同一个 Shard 下的每个副本会分散到多个节点上存储。
- ❑ **Shard**：单个 Collection 的逻辑划分，划分出来的每一份称之为 Shard，这里的划分只是逻辑概念上的分割。
- ❑ **Replica**：通过逻辑划分出来的 Shard 会以多个副本的形式实际物理存储在多个节点上，每个副本其实可以看作一个物理存在的“Core”，只不过此时副本应用的 schema.xml 和 solrconfig.xml 是托管在 Zookeeper 上的。
- ❑ **Leader**：每个 Shard 会复制出多个副本，其中一个副本会被选举为 Leader（领导），由 Leader 来负责主导分布式环境下的索引和查询请求，与 Leader 对应的还有 Follower（表示追随者）。
- ❑ **Core**：物理存在于节点硬盘上的多个索引文档集合以及这些索引文档相关的配置文件共同组成一个 Solr Core。这里的重点是 Core 中的索引文档都是物理存储在同一个节点的同个索引目录下，而 Collection 下的索引文档是以多个 Shard 下的副本分散物理存储在多个节点的索引目录下，一个副本只会物理存储于一个节点上，但同一个 Shard 下的副本必定物理存储于不同的节点上。
- ❑ **Node**：表示一个 Solr Server 实例，而一个 Solr Server 实例通常运行于 Web 容器，由于 Web 容器可以提供不同的端口号从而启动不同的 JVM 实例，这意味着同一台服务器上可以部署多个 Solr Server 实例。一个 Solr Server 实例可以拥有多个 Solr Core，而每个 Core 下可以包含某个 Collection 的部分索引文档，此时这里的每个 Core 其实就是该 Collection 某个 Shard 下的 Replica（副本）。

❑ Cluster：表示一个 Solr 集群，所有的 Solr Server 实例一起托管着所有 Solr Core，在集群环境下，每个 Solr Server 实例下管理的每个 Core 其实就是 Collection 下某个 Shard 的某一个 Replica（副本）。

为了大家能够更形象地理解 Solr 中 Collection 和 Core 两个关键性概念，我特意画了两张图，请大家认真观摩图 14-4 与图 14-5，并结合上面对 Solr 中常见术语的解释以加深理解。

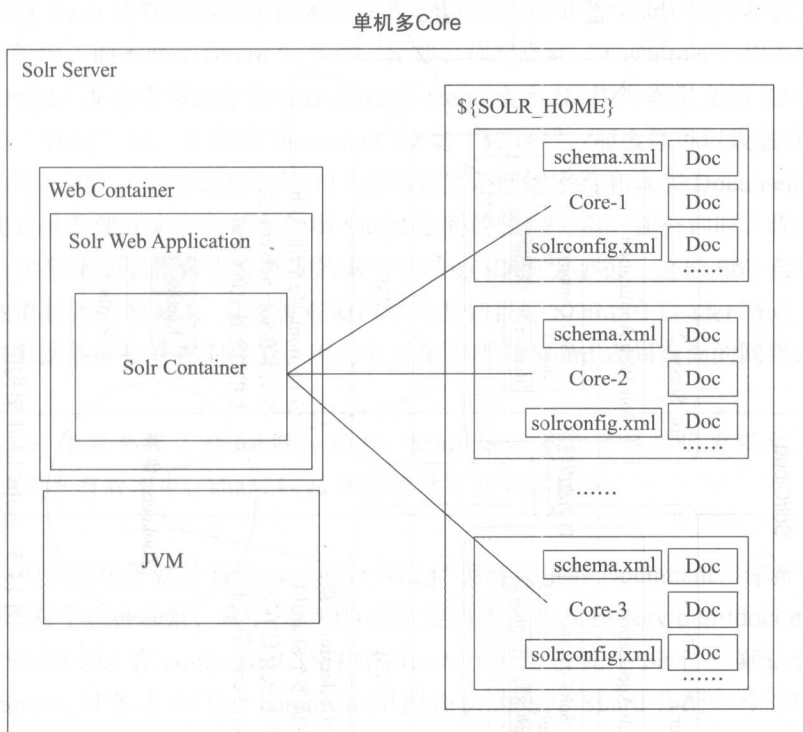


图 14-4 单机多 Core 架构图

在图 14-5 中，一个 Collection 被分成 Shard 1 和 Shard 2 两个 Shard，每个 Shard 又由两个 Replica 组成，Shard 1 的两个 Replica 分别存储于 Solr Server 1 和 Solr Server 2 两个节点上，Shard 2 同理。每个 Shard 的 Replica 会经过 Zookeeper 选举出一个 Leader，Solr Server 1 节点成为 Shard 1 的 Leader，而 Solr Server 2 节点成为 Shard 2 的 Leader。当 Shard 1 的 Leader 即 Solr Server 1 节点挂掉了，那么 Shard 1 的另一个 Replica 所在节点 Solr Server 2 会被 Zookeeper 选举为 Shard 1 新的 Leader。而 Shard 1 和 Shard 2 共同组成一个 Collection。当一个 Client 请求集群中任意一个 Solr Server 节点并将索引文档发送该节点时，该节点会首先判断当前索引文档属于哪个 Shard，确定该 Shard 的 Leader 所在 Solr Server 节点即最终目标节点，然后接收索引文档的源节点将索引文档再转发给最终目标节点，最后在该节点创建索引。

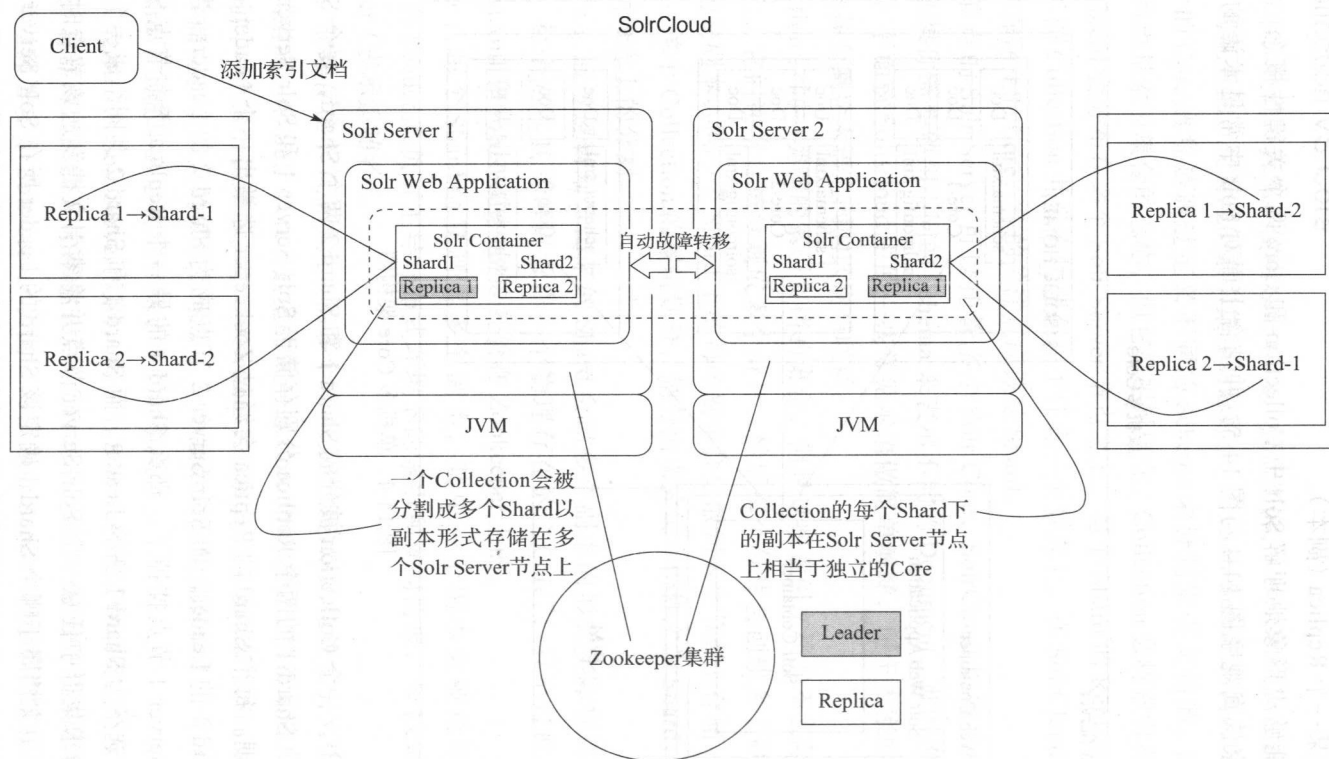


图 14-5 SolrCloud 架构图

总结: SolrCloud 下的 Collection 就好比跨多台服务器的 Multi Core (多 Core), 这里说的 Multi Core 其实就是每台服务器上的 Replica (副本)。

14.2.4 索引文档路由

Solr 允许通过指定 `router.name` 参数来设置文档路由实现, 如果使用默认的 “`compositeId`” Router (自动路由), 你可以创建并发送一个索引文档 ID 带有前缀的索引文档, 根据这个前缀进行 Hash 计算从而确定该索引文档应该发送到哪个 Shard 进行索引。这个前缀可以是任意字符串, 而不必是 Shard 的名称。比如, 你想要将 “Customer” 相关的索引文档划分为一个 Shard, 那么你可以使用 Costomer 的 name 或者 ID 作为索引文档 ID 的前缀。比如你的客户是 “IBM” 的, 而你的 Document ID 是 “12345”, 那么你可以设置你的 Document ID 为 “IBM!12345”。这里的感叹号 (!) 是用来界定前缀字符和真实 Document ID 的。这样在索引创建时就能建立索引文档与所属 Shard 之间的映射关系。在查询时, 你可以通过显式设置 `_route_` 参数来明确指定想要查询的索引文档的 ID 前缀字符, 这样 Solr 会根据你提供的前缀字符知道具体的 Shard, 直接将查询请求转发到目标 Shard 的 Leader 节点上。在某些情况下, 这种方式能够提升查询性能, 因为它避免了所有 Shard 之间查询的网络延迟。



注意 `_route_` 参数替代了 `shard.keys` 参数, `shard.keys` 参数已经标记为废弃, 在 Solr 未来的正式发布版本中, `shard.keys` 参数随时可能会被移除。

其中一种使用场景就是假如 customer “IBM” 拥有大量的 Document, 你希望将其划分为多个 Shard。对于这种场景, 索引文档 ID 的指定语法为: `shard_key/num!document_id`。其中 `/num` 表示 Shard Key 在 `compositeId` 路由的 Hash 计算中占几个 bit 位。默认情况下 `shard_key` 和 `document_id` 各占 16 位。`compositeId` 路由会为每个 Shard 分配一个 32 位的 Hash 空间, 该 Hash 空间默认均分成两段: 高位和低位, 各占 16 位。因此, “IBM/3!12345” 会使用 3 个 bit 来存储 shard key, 剩下的 29 个 bit 用来存储 Document ID。前面的 `shard_key` 用于决定 Shard 的 Hash 区间, 感叹号后面的 `document_id` 用于计算索引文档的 Hash 值, 然后根据 Hash 值确定其落入哪个 Shard。如果 `/num=2`, 则表明 `shard_key` 在 32 位 hash 空间中占 2 位, 即你可以至少创建 2^2 个 Shard, 这意味着你原来的大 Collection 可以划分为 4 份然后跨多台 Server 存储, 每台大约四分之一, 如果 `/num=3`, 相当于每台 Server 上只存储原来的大 Collection 的八分之一, 如果 `/num=4`, 那就是十六分之一, 以此类推。

在查询时, 你可以指定 `_route_` 参数或者 `shard.keys` 参数 (不过 `shard.keys` 参数已经被标记废弃不建议使用) 来明确指定在哪个 Shard 上执行查询, Solr 会自动根据该参数计算出你想要查询的 Shard 的逻辑 ID; 你也可以通过 `shards` 参数明确指定 Shard 的逻辑 ID。以下是几个 `_route_` 参数的使用示例:

```
_route_=tenant1! // 等价于 shard.keys=tenant1!, 但是 shard.keys 已不建议使用
_route_=tenant1!,tenant2! // 同时指定多个 Shard key
```

CompositeId 路由还支持两级前缀路由，比如一个前缀先按 brand（品牌）域进行路由，再按 category（分类）域进行路由：“Adidas! 鞋子!12345”、“Nike!T 恤!12345”。默认依然是 Shard Key 和 Document ID 各占 16 位，此时 Shard Key 分两级，两者再均分各占 8 位，也可以显式指定各部分占几位，比如：“Adidas/6! 鞋子 /14!12345”，即表示第一级 Shard Key 占 6 位，第二级 Shard Key 占 14 位，剩下的 Document ID 就占 $32 - (6 + 14) = 12$ 位。

当你创建了 Collection 并且明确定义了使用 Implicit Router（隐式路由），那么可以额外指定一个 router.field 参数，通过这个 field 来唯一标识每个 Document 属于哪个 Shard。如果某个索引文档的这个域缺失了，那么这个索引文档将会被拒绝。

如果用户在创建 Collection 时指定了 numShards 参数，那么系统会自动采用 compositeId 路由，否则会自动使用 Implicit 隐式路由。你可以通过 Zookeeper 上 clusterstate.json 文件中的 router 属性来查看一个 Collection 所采用的路由类型。

14.2.5 Shard 的几种状态

SolrCloud 中的 Shard 分以下几种状态：

- ❑ ACTIVE：活动状态，一个 Shard 的默认状态。
- ❑ INACTIVE：当一个 Shard 被成功分出之后，它就暂时处于 INACTIVE（非活动）状态。
- ❑ CONSTRUCTION：当一个 Shard 被重新分割，在分割处理过程中，新分出来的子 Shard 就处于 CONSTRUCTION 状态。处于该状态下的 Shard 仍然能够接收来自 Shard Leader 转发的更新请求，然而它不能参与分布式查询。
- ❑ RECOVERY：当一个 Shard 的子 Shard 需要创建 Replica 以满足 Collection 上设置的 Replica 总个数时，该 Shard 就处于 RECOVERY 状态。处于此状态的 Shard 仍然能够接收来自 Shard Leader 转发的更新请求，然而它不能参与分布式查询。

虽然对于用户而言，根本不用关心 Shard 的内在状态，但是能够熟悉 Shard 的内部状态，当 SolrCloud 出现问题时，有助于你更好地分析问题并予以解决。

14.2.6 Replica 的几种状态

SolrCloud 中的 Replica 拥有以下几种状态：

- ❑ ACTIVE（活动状态）：当一个 Replica 处于 ACTIVE 状态，即表明当前 Replica 已经准备好，可以接收更新和查询请求了。当 Replica 所在的 Solr 节点崩溃了，该 Replica 的状态信息可能还遗留在 Zookeeper 中，因此判断一个 Replica 是否真正处于 ACTIVE 状态，必须通过调用 Replica.getNodeName() 或者 ClusterState.liveNodesContain(String) 进行确认。ACTIVE 是 Replica 的默认状态。
- ❑ DOWN（宕机状态）：DOWN 是 RECOVERING 之前的第一个状态，当一个 Solr 节

点处于 DOWN 状态时，它会积极地尝试转入 RECOVERING 状态。当然一个 Solr 节点宕机之后，属于该 Solr 节点管理的 Replica 会自动进入 DOWN 状态。但是你不能完全依赖此状态，因为它并不一定可靠。

□ RECOVERING (数据恢复状态)：此状态表示 Replica 正从 Shard Leader 上执行索引恢复操作。索引恢复操作分两种：peer-sync (Update Log 增量同步) 和 full replication (索引全量恢复)。

□ RECOVERY_FAILED (数据恢复失败状态)：当 Replica 尝试进行数据恢复操作，但是尝试失败时会进入此状态。如果当前 Replica 所在 Solr 节点不在 Zookeeper 的 /live_nodes 下，那么该 Replica 的状态会被丢弃。因为它不在 Solr 集群中，记录该状态信息已经没有意义。

14.2.7 Shard 分割

当你在 SolrCloud 环境下创建一个 Collection，你需要决定初始化 Shard 的个数，然而，你很难预先知道需要几个 Shard，特别是当你的需求随时都在变化的时候。任意的分割 Shard 功能由 Solr 的 Collections API 提供，它允许你将一个 Shard 分割成两份，而原来的 Shard 不做任何改动，你可以随时在未来的某个时间点将旧 Shard 卸载。关于如何使用 Shard 的分割操作，请查阅 Solr 的 Collection API，后续章节会做介绍。

14.2.8 SolrCloud 里的自动提交

在大多数情况下，当运行在 SolrCloud 模式时，Client 请求创建索引不需要显式的 commit，此时你应该配置自动硬提交（此时 openSearcher 应该为 false）和自动软提交，使得最近更新对于查询可见。要想强制执行这种自动提交方式，需要你的所有 Client 都将索引数据发送给 SolrCloud。然而，这并不是很好执行，因此 Solr 提供了 IgnoreCommitOptimizeUpdateProcessorFactory，它允许你在 Client 应用程序里忽略显式的提交或索引优化。激活这个请求进程，需要在 solrconfig.xml 中添加如下配置：

```
<updateRequestProcessorChain name="ignore-commit-from-client" default="true">
  <processor class="solr.IgnoreCommitOptimizeUpdateProcessorFactory">
    <int name="statusCode">200</int>
  </processor>
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.DistributedUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

在上面的配置示例中，IgnoreCommitOptimizeUpdateProcessor 会返回一个 Http 200 响应状态码给 Client，然后会忽略 commit 和 optimize 请求。当然如果 Client 显式发起一个提交，你也可以返回一个 403 状态码。你还可以配置仅仅只忽略 optimize 操作，配置如下：

```
<processor class="solr.IgnoreCommitOptimizeUpdateProcessorFactory">
<str name="responseMessage">Thou shall not issue an optimize, but commits are
OK!</str>
<bool name="ignoreOptimizeOnly">true</bool>
</processor>
```

14.2.9 SolrCloud 的分布式查询请求

当一个 Solr 节点接收到一个查询请求，这个请求会被路由到被查询的 Collection 的一些 Shard 的 Replica 上。被选中的 Replica 将扮演 aggregator（聚合者）的角色，它负责创建内部请求去随机选择 Collection 的每个 Shard 上的一些 replica，协调每个 replica 的响应信息，并按照需要发起内部的子请求，比如重新改善 Facet 的值，或者请求额外的存储域，最终汇总构造一个完整的响应信息返回给 Client。

1. 分布式 Query 请求流程

当你通过 CloudSolrClient 发起一个 Solr 查询请求，首先在 requestWithRetryOnStale State 方法中会通过配置你的 zkHost 参数连接 Zookeeper，从而获取此时 Solr 集群的状态信息。其次获取用户传入的 collection 参数，如果用户没有传入 collection 参数，那么以查询请求 BaseURL 里的 collection 为准。然后传入 SolrRequest 对象和 collection 参数转到 sendRequest 方法。紧接着通过 Zookeeper 获取 Solr 集群状态信息，从 Solr 集群的状态信息中获取所有 Live Node（活跃节点）信息，因为只有对活跃节点执行查询才有意义。最后遍历所有活跃节点获取它下面的所有 Shard 信息，每个 Shard 的状态信息大致如下：

```
roles=null
leader=true
state=active
core=core_collection1_shard1_replica1
collection=collection1
node_name=ip:8984_solr
base_url=http://ip:8984/solr
```

获取每个 Shard 下的 Replica 信息并过滤掉非活跃节点下的 Replica。如果是 Update 请求，此时还会将属于 Leader 的 Replica 单独用一个 List 进行存储。对于 Update 请求而言，属于 Leader 的 Replica 始终处于所有 Replica 列表的顶端，这样做是为了保证对于 Update 请求，始终优先将请求发送给 Leader。遍历所有 Replica，首先根据每个 Replica 信息获取 node_name 即当前 Replica 所在节点的名称。用一个 Map 来存储 Replica 信息，key 为 node_name，value 为 Replica 信息。由于一个节点上可能有多个 Replica，这意味着 Map 中对于同一个节点最终只会保存一个 Replica 的信息。至于保存的是哪一个 Replica，具体取决于哪个 Replica 先遍历到。然后生成 Map 中保存的每个 Replica 的访问 BaseURL。生成规则是 shard 状态信息里的 base_url + collection，即 http://ip:8984/solr/collection1/。你将得到一个 Replica 的 CoreURL 列表，此时对 URL 列表执行 shuffle 操作即随机打乱，并将 URL 列表

交给 LBHttpSolrClient。首先会经过 LBHttpSolrClient 类的 request (Req req) 方法进行处理, 在 request (Req req) 方法内部, 会遍历 Replica 的 CoreURL 列表, 同时执行僵尸 Server 检测, 如果当前 coreURL 是僵尸 Server, 则加入僵尸 Server 队列暂时跳过不处理, 等处理完所有正常节点之后, 再开始遍历僵尸 Server 队列进行重试。如果仍然失败会开启线程池每间隔 1 分钟执行 Query("*:") 方式来检测僵尸 Server 是否重新“活过来”, 如果“活过来”了, 就将其加入 aliveServers 中。循环正常的 CoreURL 节点对齐执行 doRequest 方法, doRequest 方法内部其实只是新建了一个 HttpSolrClient 实例并调用其 request 方法真正发起了 HTTP 请求到 Solr Server。

以上说的都是 Solr Client 端的操作流程, 最终 LBHttpSolrClient 串行发送的每个 HTTP 请求会被 Solr Server 端的 SearchHandler 请求处理器所接收, SearchHandler 首先通过 distrib 参数或者 shards 参数来判断是否为分布式查询, 如果不是分布式查询就循环调用 SearchHandler 上注册的 SearchComponent (查询组件) 进行链式处理, 即下面这段代码:

```
for( SearchComponent c : components ) {
    c.prepare(rb);
}
```

对于查询来说, SearchHandler 上比较重要的一个查询组件就是 QueryComponent。其实 Solr 提供了很多查询组件, 如图 14-6 所示。

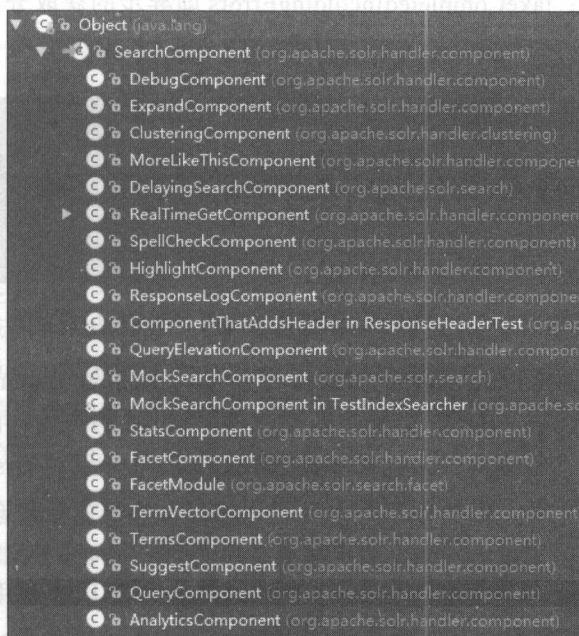


图 14-6 Solr 内置的 SearchComponent 查询组件

查询操作是交给 QueryComponent 来处理的。QueryComponent 组件的作用就是根据

传入参数构造 `QueryString`，`QParser`（查询语法解析器）根据 `QueryString` 构造出 `Query` 对象，同理还需要构造 `Filter Query`、`Group Query`、`Facet Query` 等，最终就是构造 `SolrIndexSearcher` 实例进行查询。

如果 `SearchHandler` 发现是分布式查询，同样是调用所有查询组件依次处理，不过此时调用的是查询组件的 `distributedProcess` 方法。`distributedProcess` 方法里主要是判断当初处于哪个处理阶段从而做不同的处理，这里将查询主要分为以下几个处理阶段：

```
public static int STAGE_START = 0;           // 表示查询处理开始
public static int STAGE_PARSE_QUERY = 1000; // 解析查询
public static int STAGE_TOP_GROUPS = 1500;  // 执行 Group 查询
public static int STAGE_EXECUTE_QUERY = 2000; // 执行查询
public static int STAGE_GET_FIELDS = 3000;   // 获取返回值即 fl 参数指定的域
public static int STAGE_DONE = Integer.MAX_VALUE; // 表示查询处理结束
```

接下来循环每个 `Shard` 调用 `HttpShardHandler` 的 `submit` 方法进行处理。然后循环调用 `shardHandler.takeCompletedIncludingErrors()` 异步获取响应结果。获取到响应结果后调用每个查询组件的 `handleResponses` 方法对响应信息进行数据解析。

以上说的 `HttpShardHandler` 继承自 `ShardHandler` 接口，`ShardHandler` 接口中定义了如图 14-7 所示的几个接口方法，其中比较重要的是 `prepDistributed`、`submit` 和 `takeCompletedIncludingErrors`：`prepDistributed` 方法用于执行分布式查询之前的一些准备工作；`submit` 方法即正式多线程并发执行分布式查询；`takeCompletedIncludingErrors` 即异步的获取分布式查询返回的响应信息。

```
public abstract class ShardHandler {
    public abstract void prepDistributed(ResponseBuilder rb);
    public abstract void submit(ShardRequest sreq, String shard, ModifiableSolrParams params);
    public abstract ShardResponse takeCompletedIncludingErrors();
    public abstract ShardResponse takeCompletedOrError();
    public abstract void cancelAll();
    public abstract ShardHandlerFactory getShardHandlerFactory();
}
```

图 14-7 `ShardHandler` 接口

下面我们继续看看 `submit` 方法是如何执行分布式查询的。`submit` 方法中一个重要的参数就是 `shard`。这表明 `submit` 方法仅仅是针对单个 `Shard` 的查询，而分布式查询需要的结果集是分布在多个 `Shard` 中，因此需要遍历每个 `Shard` 分别执行 `submit` 方法来获取在每个 `Shard` 上的响应结果。循环调用 `ShardHandler` 的 `submit` 方法是在 `SearchHandler` 类的 `handleRequestBody` 方法中。`submit` 方法很简单：首先获取 `Shard` 对应的访问 URL，创建一个 `Callable`，在 `Callable` 内部通过 `HttpShardHandlerFactory` 工厂类调用 `LBHttpSolrClient` 的 `request` 方法；然后将 `Callable` 放入线程池。`LBHttpSolrClient` 的 `request` 方法内就是简单的创建 `HttpSolrClient` 实例去循环访问每个 `Shard` URL。

在 `submit` 方法中有个 `getURL` 方法需要引起大家注意，在 `getURL` 方法中会调用 `prefer`

CurrentHostForDistributedReq 方法, 对于 preferCurrentHostForDistributedReq 方法而言, 如果用户传递了 preferLocalShards 参数且为 true, HttpShardHandler 会将 URL 列表中与当前接收请求的主机 ip 和端口一致的 URL 提到列表顶端, 即当前主机优先加载本地的 Shard 进行查询。为了便于理解整个查询执行流程, 请看如图 14-8 所示的查询流程图。

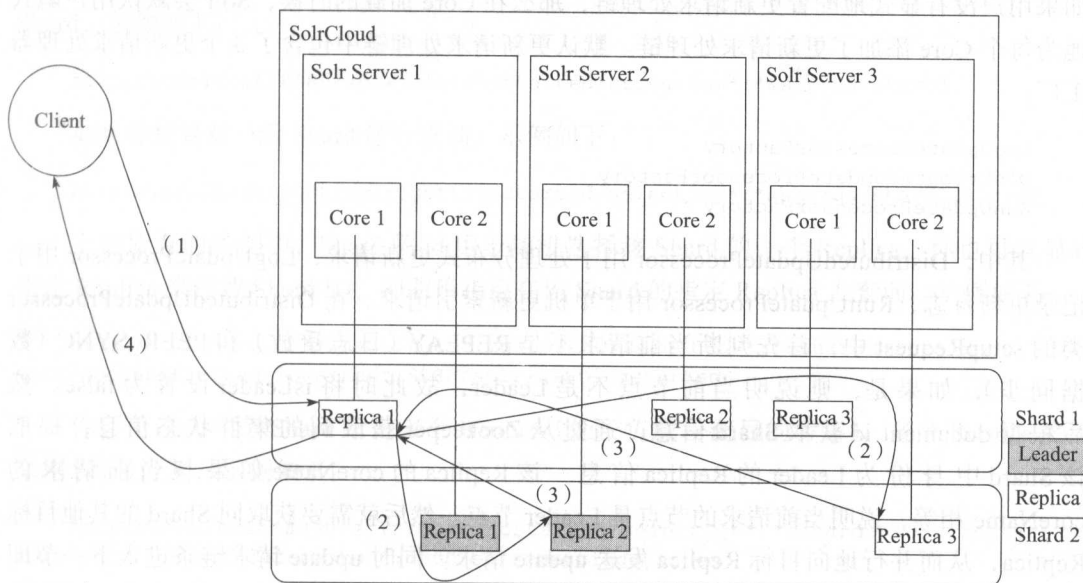


图 14-8 Solr 分布式查询执行流程示意图

在图 14-8 中, 第 (1) 步 Solr 客户端用户通过 CloudSolrClient 向 SolrCloud 集群发送查询请求, 请求先随机到图中 Solr Server 1 节点的 Replicat1, 然后请求会被 Solr Server 1 节点上的 SearchHandler 所接收, 判断是否为分布式查询。显然此时是分布式请求, 因此 SearchHandler 会遍历每个 Shard 并构造 N 个非分布式查询请求 (注意, 这里的 N 由 Shard 的个数决定)。然后交由 HttpShardHandler 处理, HttpShardHandler 内部会新建一个 Callable 任务并放入线程池进行异步处理。Callable 内部会通过 HttpSolrClient 去请求其他 Solr Server 节点上的 Replica, 即图中的第 (2) 步。紧接着每个 Replica 会异步返回响应信息给 Replicat1, 它充当着响应信息聚合的角色, 即图中的第 (3) 步, 最后 Replicat1 汇总响应信息返回给 Client, 即第 (4) 步。

2. 分布式 Update 请求流程

说到分布式查询流程, 顺带说一说分布式 Update 请求的流程。首先当你使用 CloudSolrClient 构造一个 Update 请求时, 内部会首先判断是否为 Update 请求, 如果是, 那么就转到 directUpdate 方法进行处理。directUpdate 方法内部会首先根据 Document 的 ID 进行文档路由, 确定该 Document 应该发往哪个 Shard, 然后获取该 Shard 下的所有 Replica 的 CoreURL, 找出每个 Shard 的 Leader 的 CoreURL, 这样每个 Document (因为可能为批量

更新)就对应了一个 Leader 访问 URL,为每个 Leader URL 构造一个 Callable 任务并放入线程池异步执行。Callable 内部会使用 LBHttpSolrClient 请求每个 Document 对应的 Leader 节点。然后 update 请求被 UpdateRequestHandler 所接收处理,在 UpdateRequestHandler 的 handleRequestBody 方法里,会首先获取 UpdateRequestProcessorChain (更新请求处理链),如果用户没有显式地配置更新请求处理链,那么在 Core 加载的时候,Solr 会默认用户隐式地为每个 Core 添加了更新请求处理链。默认更新请求处理链中包含了 3 个更新请求处理器工厂:

```
LogUpdateProcessorFactory
DistributedUpdateProcessorFactory
RunUpdateProcessorFactory
```

其中,DistributedUpdateProcessor 用于处理分布式更新请求,LogUpdateProcessor 用于记录更新日志,RunUpdateProcessor 用于单机更新索引请求。在 DistributedUpdateProcessor 类的 setupRequest 中,首先判断当前请求不是 REPLAY (日志重放)和 PEER_SYNC (数据同步),如果是,则说明当前节点不是 Leader,故此时将 isLeader 设置为 false。然后根据 document id 获取 Shard 信息,通过从 Zookeeper 获取到的集群状态信息,提取该 Shard 中身份为 Leader 的 Replica 信息。该 Replica 的 coreName 如果与当前请求的 coreName 相等,说明当前请求的节点是 Leader 节点。然后就需要获取同 Shard 的其他目标 Replica,从而并行地向目标 Replica 发送 update 请求。同时 update 请求链条进入下一节即 RunUpdateProcessor,RunUpdateProcessor 会对当前 Leader 节点执行索引更新操作以及记录 Update log。如果当前节点是从 Leader 节点转发过来的,说明当前 Replica 节点不是 Leader,此时会判断当前 Replica 节点所属 Shard 是否与 Document ID 路由计算出来的 Shard 保持一致,如果不一致,就将索引更新请求再转发到 Document 所属的 shard Leader 节点上。如果一致,就进入 RunUpdateProcessor 执行索引更新处理。最终 Document 会在该 Shard 的所有 Replica 节点上被更新,从而保持数据同步。总之,对于 Update 请求,Client 端会尽力找出当前需要更新的 Document 所属的 Leader,如果找出来了,就直接发往目标 Leader,这样会减少在其他 Shard Leader 之间转发带来的网络延迟。但是如果 Client 端确定不了 Shard Leader,那么会发送给所有 Replica,由 Solr Server 端去判断。Solr Server 端判断当前节点是否为 Leader,如果是 Leader 就本地更新索引文档并记录 Update long (便于数据恢复),同时转发给同 Shard 的其他 Replica。其他 Replica 收到 Leader 转发的 update 请求时,会判断该 Leader 是否是自己所属 Shard 的 Leader:如果是,那么直接本地更新索引文档(此时不写 Update Log);如果不是,就转发到 Document 真正所属的 Shard Leader,由该 Leader 去处理。

3. 指定查询的目标 Shard

使用 SolrCloud 的其中一个好处就是你可以对一个分布在多个 Shard 上的很大的 Collection 进行查询。比如,有些时候你可能只对某些 Shard 上的索引文档感兴趣,通过

SolrCloud, 你可以选择是在所有的索引数据上执行查询还是仅仅只是部分 Shard 上。在 Collection 的所有 Shard 上执行查询操类似下面这样, 看起来 SolrCloud 似乎没起作用一样:

```
http://localhost:8080/solr/gettingstarted/select?q=*:*
```

如果你只想要在某一个 Shard 上查询, 可以通过该 Shard 的逻辑 ID 来指定想要查询的 Shard, 示例如下:

```
http://localhost:8080/solr/gettingstarted/select?q=*:*&shards=shard1
```

如果你想要对一组 Shard 进行查询, 示例如下:

```
http://localhost:8080/solr/gettingstarted/select?q=*:*&shards=shard1,shard2
```

上面所有的实例中, Shard 的 id 用于随机选择该 Shard 的一个 Replica。你也可以显式指定 Replica 来代替 Shard ID, 即明确指定在该 Shard 的指定 Replica 上查询, 示例如下:

```
http://localhost:8080/solr/gettingstarted/select?q=*:*&shards=localhost:7574/  
solr/gettingstarted,localhost:8080/solr/gettingstarted
```

你还可以为一个 Shard 指定多个 Replica (处于负载均衡目的考虑), 多个 Replica 之间采用 | 符号分割, 示例如下:

```
http://localhost:8080/solr/gettingstarted/select?q=*:*&shards=localhost:7574/  
solr/gettingstarted|localhost:7500/solr/gettingstarted
```

当然, 上面的方式可以混合使用, 示例如下:

```
http://localhost:8080/solr/gettingstarted/select?q=*:*&shards=shard1,localho  
st:7574/solr/gettingstarted|localhost:7500/solr/gettingstarted
```

此外, 可以对多个 Collection 进行查询, 示例如下:

```
http://localhost:8080/solr/collection1/select?collection=collection1,collection  
2,collection3
```

4. 配置 shardHandlerFactory

你可以为分布式查询指定一些并发性和线程池相关的配置, 细粒度地控制并调节它以满足特定的需求。solrconfig.xml 中 shardHandler 的配置示例如下:

```
<requestHandler name="standard" class="solr.SearchHandler" default="true">  
<!-- shardHandler 配置参数 -->  
<shardHandler class="HttpShardHandlerFactory">  
<int name="socketTimeout">1000</int>  
<int name="connTimeout">5000</int>  
</shardHandler>  
</requestHandler>
```

表 14-1 列举了 shardHandler 支持的所有配置参数。

表 14-1 shardHandler 配置参数表

参数名	描 述	默认值
socketTimeout	socket 等待超时时间 (单位: 毫秒)	0 (使用操作系统默认值)
connTimeout	将一个 Http 连接绑定到一个 socket 上允许的最大超时时间 (单位: 毫秒)	0 (使用操作系统默认值)
maxConnectionsPerHost	最大的并发连接数, 针对每个单独 Shard 设置	20
maxConnections	分布式查询中, 最大的并发连接数	10 000
corePoolSize	初始化线程池中的线程数量	0
maximumPoolSize	线程池中线程的最大数量	Integer.MAX_VALUE
maxThreadIdleTime	设置线程在被回收之前空闲的最大时间 (单位: 秒)	5 (秒)
sizeOfQueue	如果指定此参数, 那么线程池会使用队列来代替直接缓冲区, 对于追求高吞吐量的系统而言, 可能希望配置为 -1 即使用直接缓冲区, 而对于追求低延迟的系统可能希望配置一个合理的队列大小来处理请求。	-1
fairnessPolicy	用于选择 JVM 中特定的用于队列的公平策略: 如果启用公平策略, 分布式查询会以先进先出的方式来处理请求, 但是是以有损吞吐量代价; 如果禁用公平策略, 会提高分布式查询吞吐量, 但是是以响应延迟为代价	false
useRetries	是否启用 HTTP 连接自动重试机制, 避免由于 HTTP 连接池的限制或者竞争导致的 IOException, 默认未开启	false

5. 配置分布式的 IDF

在相关性评分计算中需要进行 Document 和 Term 的统计, Solr 提供了 4 种开箱即用的关于 Document 统计的实现:

- ❑ LocalStatsCache: 这种方式仅仅只使用本地的 Document 和 Term 统计来计算相关性, 当 Term 跨多个 Shard 分布, 可考虑这个实现。当你没有配置 <statsCache>, 默认就是这个实现。
- ❑ ExactStatsCache: 这个实现会使用全局的值 (跨多个 Collection) 来进行文档频率计算。
- ❑ ExactSharedStatsCache: 它在功能上与 ExactStatsCache 很相似, 但是 ExactStatsCache 的全局统计对于相同 Term 在子请求中可以被重用。
- ❑ LRUSStatsCache: 这个实现使用 LRU 缓存来缓存全局统计, 然后会在各个请求之间共享。

这些实现可以在 solrconfig.xml 中通过 <statsCache> 元素进行配置, 示例如下:

```
<statsCache class="org.apache.solr.search.stats.ExactStatsCache"/>
```

6. 分布式查询死锁问题

每个 Shard 均服务于顶级查询请求, 然后为其他 Shard 创建子请求, 你必须确保能提供

HTTP 请求的最大线程数大于可能的最大请求数，否则可能会导致分布式死锁。比如，在两个 Shard 的情况下可能会发生死锁，每个 Shard 使用单线程服务于 HTTP 请求，这些线程都可以并发接收顶级请求，且为其他 Shard 创建子请求，因为已经没有剩余的线程用于服务请求，接下来的请求将会被阻塞，直到其他待处理的请求被处理完成（但是它们不可能完成，因为它们在等待子请求）。你可以为 Solr 提供足够的线程来避免类似的死锁情况。

7. 优先加载本地 Shard

Solr 允许传递一个可选的 boolean 参数，`preferLocalShards`，来指示分布式查询应该优先加载本地的 Replica。换句话说，如果一个查询包含 `preferLocalShards=true`，那么这个查询控制器会优先查询本地的 Replica 来服务于查询，而不用跨集群随机选择 Replica 的方式。这个功能很适用于当一个查询请求为每个索引文档返回很多 Field 或者大文本域时，因为它避免了在网络间传输大量数据。此外，它能将有问题问题的 Replica 对性能的影响降低到最小，因为减小了被降级的 Replica 被其他正常 Replica 命中的可能性。

最后需要注意的是，此选项只能在负载均衡请求中使用，比如使用 `CloudSolrClient`，否则可能会导致集群热点问题，因为请求不能均衡地在集群中分布。

14.2.10 读写端的自动容错

SolrCloud 在读写操作上支持弹性、高可用性、自动容错性，这意味着，Read 操作随时都可以返回数据，尽管有些节点挂掉了，但写操作不会丢失数据。

1. 读操作的自动容错

在 SolrCloud 集群中，每个独立的节点会跨 Collection 下的所有 Replica 对读请求进行负载均衡。你在集群外部仍然需要一个负载均衡器来与集群通信，或者需要一个智能 Client 能理解如何读取和操作存储于 Zookeeper 上 Solr 的 metadata（元数据信息），并且请求 Zookeeper 发现请求应该发送到哪个节点上。为此，SolrJ 提供了 `CloudSolrClient`。即便集群中某些节点离线不可用了，只要每个 Shard 至少还有一份 Replica 存在，集群中的其他节点就会正常响应查询请求。Shard 的 Replica 个数越多，当出现节点故障时，集群能够继续处理查询请求的可能性越大。

只要一个 Solr 节点能够与每个 Shard 的至少一个 Replica 通信，它就能够返回查询结果（即便它不能与 Zookeeper 进行通信）。但是一个节点若不能与 Zookeeper 进行通信，假如一个 Collection 的结构发生重大变化，比如添加 / 删除了一个 Shard 或者一个 Shard 重新分割了子 Shard，则可能会导致返回的数据不是最新的或者不正确的数据。在每个查询响应信息中都会包含一个 `zkConnected` 请求头信息，用于指示该节点在查询时能够与 Zookeeper 进行连接。

如果有多个 Shard 不可用了，那么 Solr 默认会返回查询失败。然而，很多时候可以接收部分结果集，所以 Solr 提供了 `shards.tolerant` 参数（默认值 `false`），如果 `shards.tolerant=true`，

那么查询将可以返回部分结果集，且 Solr 会在响应头信息中通过 `partialResults` 来提醒 Client，当前返回的是不是部分结果集。

2. 写操作的自动容错

当你向集群中的任意节点发送一个 Update 请求时，该节点会首先判断当前节点上是否是该 Shard 的 Leader，如果不是，会将请求转发到该 Leader 所在节点上。使用 version 来确保每个 Replica 都是最新的，如果 Leader 挂掉了，其他 Replica 会顶替上，这种架构允许在发生故障或灾难时恢复你的数据，即便你采用的是软提交。

每个节点都会创建 Transaction Log（事务日志），也就是说，内容的每一步更新都会被记录。事务日志用于决定当前节点上的内容应该包含在哪个 Replica 里。当一个新的 Replica 被创建，它会从 Leader 同步数据并且根据事务日志确定哪个更新应该包含在当前 Replica 中。如果数据同步操作失败了，它会根据事务日志进行重试，因为事务日志是由更新记录组成，它能增加索引操作的健壮性，因为当索引操作突然中断时，它支持重做未提交的更新。如果一个 Leader 挂掉了，会从其他 Replica 选举一个新的 Leader，然后由新的 Leader 向其他 Replica 发起一个同步操作，如果同步操作成功，那么所有 Replica 数据将保持一致。如果一个 Replica 脱离太久无法同步，Leader 会向该 Replica 发送 Recovery（数据恢复）请求，强迫该 Replica 主动进行数据恢复。如果由于 Core 重载而导致更新操作失败，Leader 会通知其他节点更新失败，并启动 Recovery 过程。

Solr 中的 Recovery 操作分为 Peer sync 和 Replication 两种方式：

- ❑ Peer sync：如果需要 Recovery 的节点中断的时间较短，只是丢失少量的 Update 请求，那么它可以从 Leader 的 Update log 中获取。这个临界值是 100 个 update 请求，如果大于 100 个，该 Replica 节点就会从 Leader 进行完整的索引快照恢复。
- ❑ Replication：如果该节点下线太久，以至于不能从 Leader 进行同步，它就会以 HTTP 的形式进行完整的索引快照恢复。

当使用 Replica Factor 大于 1 时，一个 Update 请求可能会在 Leader 上执行成功而在其他一个或多个 Replica 上执行失败。例如，一个 Collection 有 1 个 Shard 和 3 个 Replica Factor。此时，你有一个 Shard Leader 和 2 个 Replica，如果一个 Update 请求只是在 Leader 上执行成功，不管出于什么原因，更新请求仍然会被认为执行成功，当执行失败的 Replica 恢复时会自动与 Leader 进行数据同步。Solr 支持为 Update 请求设置可选的 `min_rf` 参数，同时会在响应头信息里返回实际执行成功的 Replica 节点个数。Client 并不能强制要求在指定个数的 Replica 上执行成功，且 Solr 无法回滚已经执行成功的 Replica 节点，因此，Solr 只能告诉 Client 有几个 Replica 更新成功了。

在 Client 端，如果接收到的 Replica Factor 小于可接受的级别，Client 应用程序可以采取额外的措施来进行降级处理。例如，在一个 Collection 被降级期间，Client 应用程序想要记录更新请求日志，以便当问题解决了之后能重新发送更新请求，简而言之，`min_rf` 参数用

于指示 Client 端在一个 Collection 降级期间一个更新请求被接受了。

14.2.11 Zookeeper

ZooKeeper 是分布式系统里的协调框架，Solr 使用 ZooKeeper 来实现 3 个关键操作：

- 配置文件的集中存储与分发；
- 探测和通知集群状态更新；
- Shard Leader 选举。

SolrCloud 选择使用 ZooKeeper 是因为它比较成熟、稳定，并且被大量复杂的分布式系统所广泛使用。本节着重讲解关于 ZooKeeper 的一些重要概念。

1. Zookeeper 的数据模型

ZooKeeper 采用类似文件系统的分层次命名空间方式来组织数据，每一个层级称为“Znode”，每个 Znode 包含了一些基本的元数据信息，比如数据版本号、创建时间、最后一次更新时间，同时存储少量的数据。Znode 并不是为了存储大数据对象而设计的，每个 Znode 默认最大可存储 1MB 的数据，因为出于性能考虑，Znode 需要缓存在内存中。ZooKeeper 也不是通用的数据存储系统，它更适合用于存储小块的元数据信息。

ZooKeeper 中的 Znode 大致有 4 种类型，在 `org.apache.zookeeper.CreateMode` 类中定义，如下所示：

- EPHEMERAL：临时节点。
- PERSISTENT：持久化节点。
- EPHEMERAL_SEQUENTIAL：带序号的临时节点。
- PERSISTENT_SEQUENTIAL：带序号的持久化节点。

Znode 的类型在创建之后就不能再修改。临时 Znode 在客户端会话结束之后会被删除，临时节点不能再有子节点。持久化 Znode 当客户端会话结束时也不会被删除，除非客户端显式地请求删除该 Znode。在创建 Znode 时你还可以为节点指定序号，序号在父节点要保持唯一。序号采用 10 位数字表示，不够位数则采用前导零补充，Znode 的序号最大值为 232-1。带序号的 Znode 节点可以用于实现共享锁和分布式队列。

可以监控 Znode，包括这个目录节点中存储数据的修改，子节点目录的变化等，一旦发生变化可以通知设置了 Watcher 的客户端。这个功能是 Zookeeper 最重要的特性，通过这个特性可以实现配置的集中管理，集群管理，分布式锁等功能。

每个 Znode 上除了保存可以被用户 CRUD 的数据之外，还保存了当前节点的 ACL 信息（访问权限数据）。ACL 信息决定了哪些用户可以对当前节点执行哪些操作。

2. Zookeeper 中的 Znode Watcher

Zookeeper 的另一个核心概念就是 Znode 的 Watcher（监视器）。任何一个 Client 应用程序都可以将自己注册为一个 Znode 的 Watcher。如果 Znode 的状态发生改变，ZooKeeper 会

通知所有被注册的 Watcher。例如，Solr 在 /clusterstate.json 节点上注册了一个 Watcher，这样当集群状态发生改变时（比如某个 Replica 挂掉了），Solr 就会接收到 Zookeeper 的通知。

3. 产品环境下 Zookeeper 配置建议

默认 SolrCloud 是在同一个进程内启动一个嵌入式的 ZooKeeper 实例，但是，在产品环境下，强烈建议单独部署 Zookeeper 集群（至少 3 个节点）。如果 Zookeeper 部署 3 个节点，这样当一个节点时并不需要停机维护。但是如果你的 ZooKeeper 整个挂掉，Solr 可以继续接收查询请求，但是不能接收更新请求，这是一个防护机制。此时 Solr 能响应查询是因为每个节点缓存了集群状态。在你启动 SolrCloud 时可以指定 zkHost 参数来明确告诉 Solr：ZooKeeper 集群各节点的访问 ip 和端口，示例如下：

```
-DzkHost=zk1.example.com:2181,zk2.example.com:2181,zk3.example.com:2181
```

另一个需要理解的参数就是 ZooKeeper 客户端超时时间。当一个 Solr 节点加入集群，它会首先创建一个 Znode 并指示自己是一个 Live Node（活跃节点）。如果一个 Solr 节点崩溃了，ZooKeeper 会在到达设置的超时时间之后开始探测节点崩溃。默认超时时间是 15 秒，你可以通过 zkClientTimeout 进行重新设置，示例如下：

```
-DzkClientTimeout=30000
```

当 JVM 执行 Full GC 时会暂停所有执行线程，包括缓存 Zookeeper 会话的线程。因此，如果 Full GC 占用的时间比 Zookeeper 客户端的超时时间还长，那么 Zookeeper 会出现离线状态。当 Full GC 执行完毕，Solr 会尝试重新与 Zookeeper 建立连接。如果你发现 Solr 节点纷纷失去与 Zookeeper 的连接，那么此时需要开启 JVM GC 日志，查看是否 Full GC 活动占用了大量时间。你可以启用 CMS 垃圾回收器，开启多线程垃圾回收，以及增大 SurvivorRatio（幸存区）比例，延长对象存活时间，防止过早进入。

4. 配置集中管理与分发

Zookeeper 另一个强大功能就是配置集中管理和分发。当在 SolrCloud 集群中创建一个新的 Collection 时，你需要将 solrconfig.xml 和 schema.xml 等配置文件上传到 Zookeeper，这样当 SolrCloud 运行时，会自动从 Zookeeper 拉取配置文件。如果需要修改 solrconfig.xml 配置，你只需要上传更新后的配置到 Zookeeper，然后重新加载你的 Collection，从而触发集群中的所有节点从 Zookeeper 中更新最新的配置。借助这个强大的特性，你可以在任意时刻往集群中添加一个节点（该节点将会应用到最新的配置文件）。想象一下，假如你管理着 100 台服务器，你需要修改 solrconfig.xml，当修改完毕之后，你需要手动 push 到其他所有节点，然后该 Collection 需要重新加载，但是现在你只需要 push 到 Zookeeper，其他节点会自动拉取最新配置。

现在你应该已经对 Zookeeper 的概念有了基本的理解，下面让我们开始动手搭建 Zookeeper 集群。

5. Zookeeper 集群搭建

这里以 Zookeeper 3.4.5 版本为例，在 CentOS 6.5 环境下对搭建过程进行讲解说明，首先你需要访问 Zookeeper 官网下载 Zookeeper 安装包，Zookeeper 的官方下载地址如下所示：

`http://archive.apache.org/dist/zookeeper/`

然后需要安装 JDK 1.6 +，至于如何安装 JDK，这里就不展开了。这里我以 3 台虚拟机进行模拟，3 台机器的域名依次是 `linux.yida01.com`、`linux.yida02.com`、`linux.yida03.com`。你需要对 3 台服务器设置时间同步，这在分布式系统环境中非常重要。

(1) 选择其中一台作为 ntp 时间服务器

检查 ntpd 服务器是否启动。

```
service ntp restart
```

如果提示 `ntpd: unrecognized service` 未发现 ntpd 服务，则表明当前机器尚未安装 ntpd 服务，则需要先安装 ntpd，ntpd 服务安装命令：`yum-y install ntp`。ntpd 服务安装成功后，请启动 ntpd 服务：`sudo service ntpd start`。设置 ntpd 服务随开机启动：

```
sudo chkconfig ntpd on
sudo chkconfig --list | grep "ntpd"
```

你需要修改 `/etc/ntp.conf` 配置文件：

1) 去掉 `restrict 192.168.1.0 mask 255.255.255.0 nomodify notrap` 注释，同时修改 IP 与你的主机 IP 在同一网段。

2) 注释掉外部 ntp 时间服务器。

```
#server 0.centos.pool.ntp.org iburst
#server 1.centos.pool.ntp.org iburst
#server 2.centos.pool.ntp.org iburst
#server 3.centos.pool.ntp.org iburst
```

3) 添加以下两行内容。

```
server 127.127.1.0 # local clock 即本地时钟同步
fudge 127.127.1.0 stratum 10
```

4) 重启你的 ntpd 服务。

```
sudo service ntpd restart
```

(2) 其他服务器与 ntp 时间服务器进行时间同步

在其他服务器上通过 `ntpdate` 命令与 ntp 时间服务器进行时间同步。

```
ntpdate -u linux.yida01.com
```

`linux.yida01.com` 即你的 ntpd 服务器的域名，如果指定 `ntpdate` 命令提示 `ntpdate: command`

not found, 则说明你当前服务器尚未安装 ntpdate, 请使用 yum 命令进行安装, 如下:

```
yum -y install ntpdate
```

时间同步的时候记得先关闭当前服务器的防火墙:

```
service iptables stop
```

同理, 在其他服务器上执行 ntpdate-u 进行时间同步。

(3) 编写时间同步定时任务

如果服务器数目比较多, 不可能手动地去每台服务器上执行 ntpdate-u 命令进行时间同步, 这时候就需要编写一个 crontab 定时任务: 每 10 分钟自动进行时间同步。

```
crontab -e
0-59/10 * * * * /usr/bin/sudo /usr/sbin/ntpdate -u linux.yida01.com
```



注意 除了 ntp 服务器 (linux.yida01.com) 以外的每台服务器上都要添加此 crontab 定时任务。

执行 sudo 经常会碰到 “sudo: sorry, you must have a tty to run sudo” 这个情况, 这时只需修改一下 sudo 的配置就好了。

```
vi /etc/sudoers (最好用 visudo 命令)
```

注释掉 “Default requiretty” 一行, 如下:

```
#Default requiretty
```

意思就是 sudo 默认需要 tty 终端。注释掉就可以在后台执行了。

(4) 设置 BIOS 硬件时钟与本地时钟同步

这里介绍两种方式。

第一种: 通过 linux 命令手动同步。

```
sudo hwclock -s           // 让硬件时钟与系统本地时钟保持同步
sudo hwclock --show       // 显式当前硬件时钟时间
```

第二种: 通过修改 /etc/sysconfig/ntpd 配置文件。

```
sudo vi /etc/sysconfig/ntpd
```

在文件开头添加以下内容:

```
SYNC_HWCLOCK=yes
```



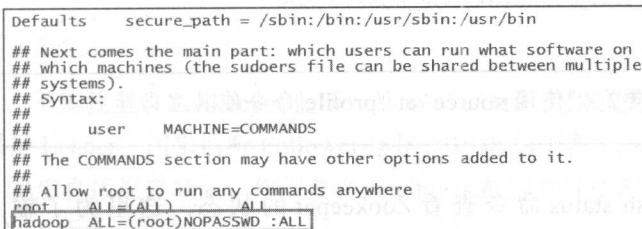
注意 不管使用哪种方式设置, 集群中的每台服务器都要进行硬件时钟与系统本地时钟的同步工作。

然后请将 Zookeeper 安装包上传到 /opt/software 目录下，这里以 /opt/software 目录为例，你可以上传到其他任意你想要放置的目录。接下来在 /opt/modules 目录下创建目录 zookeeper，并更改其所属用户和用户组为 hadoop 用户。

```
sudo chown -R hadoop:hadoop zookeeper/
```

这里的 hadoop 用户为新创建的具有 sudo 权限的用户（具体设置如图 14-9 所示），当然也可以直接使用 Root 用户，但是一般不建议直接使用 Root 用户操作。你可以创建一个用户，步骤如下：

```
useradd hadoop
passwd hadoop
给 hadoop 用户设置 sudo 权限且可以免密码登录
vi /etc/sudoers
```



```
Defaults    secure_path = /sbin:/bin:/usr/sbin:/usr/bin
## Next comes the main part: which users can run what software on
## which machines (the sudoers file can be shared between multiple
## systems).
## Syntax:
##
##      user    MACHINE=COMMANDS
##
## The COMMANDS section may have other options added to it.
##
## Allow root to run any commands anywhere
root    ALL=(ALL)    ALL
hadoop  ALL=(root)NOPASSWD :ALL
```

图 14-9 为普通用户设置 sudo 权限

当然，这里的用户名称和密码可以修改成其他。紧接着请解压 zookeeper 安装包至 /opt/modules/zookeeper 目录下，命令如下所示：

```
tar -zxvf /opt/software/zookeeper-3.4.5.tar.gz -C /opt/modules/zookeeper/
```

进入 zookeeper 的 conf 目录，复制 zoo_sample.cfg 文件并重命名为 zoo.cfg，命令如下所示：

```
cp zoo_sample.cfg zoo.cfg
```

创建 zookeeper 的数据目录 /opt/modules/zookeeper/zookeeper-3.4.5/tmp/data，命令如下所示：

```
cd /opt/modules/zookeeper/zookeeper-3.4.5/
mkdir -p tmp/data
```

然后编辑 zoo.cfg 配置文件，修改 zookeeper 的 dataDir，如下所示：

```
cd /opt/modules/zookeeper/zookeeper-3.4.5/conf
vi zoo.cfg
dataDir=/opt/modules/zookeeper/zookeeper-3.4.5/tmp/data/
```

配置 Zookeeper 环境变量，如下所示：

```
sudo vi /etc/profile
#JDK
```

```

JAVA_HOME=/opt/modules/jdk1.7.0_67
JRE_HOME=/opt/modules/jdk1.7.0_67/jre
#Maven
MAVEN_HOME=/opt/modules/apache-maven-3.0.5
#Hadoop
HADOOP_HOME=/opt/modules/hadoop/hadoop-2.5.0
#Zookeeper
ZOOKEEPER_HOME=/opt/modules/zookeeper/zookeeper-3.4.5
#protobuf
PROTOBUF_HOME=/usr/local/protobuf
PATH=$PATH:$JAVA_HOME/bin:$JRE_HOME/bin:$MAVEN_HOME/bin:$PROTOBUF_HOME/
bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$ZOOKEEPER_HOME/bin
CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar:$JRE_HOME/lib
export JAVA_HOME JRE_HOME MAVEN_HOME PATH CLASSPATH PROTOBUF_HOME ZOOKEEPER_
HOME
export PKG_CONFIG_PATH=$PROTOBUF_HOME/lib/pkgconfig/
export HADOOP_LOG_DIR=${HADOOP_HOME}/logs

```



注意 修改完请不要忘记使用 `source /etc/profile` 命令使其立即生效。

使用 `zkServer.sh status` 命令查看 Zookeeper 的状态，这里为了输入命令方便，为 `zkServer.sh` 创建了一个 `zkServer` 的别名，而不用每次都加 `.sh` 后缀，如下所示：

```
alias zkServer="zkServer.sh"
```

执行以下命令，以本地模式启动 Zookeeper：

```

zkServer start
zkServer status           // 执行这条命令后，你会看到提示信息：Mode: standalone

```

到这一步 Zookeeper 的本地模式就搭建成功了，通过 `zkCli.sh` 脚本打开 Zookeeper 客户端进行验证。这里为了方便操作，依然为 `zkCli.sh` 脚本创建了一个别名 `"zkcli"`，如下所示：

```

alias zkcli="zkCli.sh"    // 为 zkCli.sh 脚本创建一个别名
zkcli                    // 使用 zkCli.sh 脚本连接 Zookeeper Server 进行连接测试

```

下面是一个 Zookeeper 的简单操作示例，如图 14-10 所示。

最后你可以输入 `quit` 命令退出 zookeeper 客户端。接下来请随我继续配置 Zookeeper 的集群模式。首先关闭 zookeeper 服务：

```
zkServer stop
```

然后编辑 `zoo.cfg` 配置文件，配置示例如下所示：

```

server.1=linux.yida01.com:2888:3888
server.2=linux.yida02.com:2888:3888
server.3=linux.yida03.com:2888:3888

```

```
[zk: localhost:2181(CONNECTED) 5] create /hello hello 创建一个znode
Created /hello
[zk: localhost:2181(CONNECTED) 6] get /hello 获取一个znode的值
hello
cZxid = 0x2
ctime = Fri Jun 03 10:29:52 CST 2016
mZxid = 0x2
mtime = Fri Jun 03 10:29:52 CST 2016
pZxid = 0x2
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
[zk: localhost:2181(CONNECTED) 7] ls /hello
[]
[zk: localhost:2181(CONNECTED) 8] ls / 查看znode的节点信息
[hello, zookeeper]
[zk: localhost:2181(CONNECTED) 9]
```

图 14-10 Zookeeper 客户端命令操作示例图


格式如下：

server. 序号 = 域名或 ip:2888:3888

序号 1 ~ 255，不能重复。Zookeeper 监听三个端口：2181 用来监听 Zookeeper 客户端连接；2888，如果是 Leader，用来监听 Follower 连接；3888 用于在 Leader 选举阶段监听其他服务器的连接。要想直接配置域名，你需要在 /etc/hosts 配置文件中配置 IP 地址与域名的映射关系，如图 14-11 所示。

```
[hadoop@linux conf]$ cat /etc/hosts
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4
::1 localhost localhost.localdomain localhost6 localhost6.localdomain6
192.168.133.100 linux.yida01.com
192.168.133.101 linux.yida02.com
192.168.133.102 linux.yida03.com
```

图 14-11 IP 与域名映射

 **注意** 如果你想要使用域名访问，那么集群中的每台机器都需要在 /etc/hosts 中配置 IP 与域名的映射。

紧接着，请在 zookeeper 的 dataDir 目录下创建 myid 文件，具体操作如图 14-12 所示。

```
[hadoop@linux zookeeper-3.4.5]$ cd tmp/data
[hadoop@linux data]$ touch myid
[hadoop@linux data]$ echo 1 > myid
[hadoop@linux data]$ cat myid
1
[hadoop@linux data]$
```

图 14-12 为 Zookeeper 创建 myid 文件

然后将当前这台服务器上安装好的 Zookeeper 通过 scp 命令分发到其他服务器上。不过请注意：分发之前，你必须在其他服务器上提前创建好 /opt/modules/zookeeper 目录，且设

置 zookeeper 目录的所属者和所属用户组为当前 hadoop 用户，操作命令如下所示：

```
cd /opt/modules
sudo mkdir zookeeper
sudo chown -R hadoop:hadoop zookeeper/
```

然后将 linux.yida01.com 这台机器上安装好的 zookeeper 通过 scp 命令分发到 linux.yida02.com 这台机器的 /opt/modules/zookeeper/ 目录下，在 linux.yida01.com 这台服务器上执行如下命令：

```
scp -r /opt/modules/zookeeper/zookeeper-3.4.5 linux.yida02.com:/opt/modules/zookeeper/
```

同理执行：

```
scp -r /opt/modules/zookeeper/zookeeper-3.4.5 linux.yida03.com:/opt/modules/zookeeper/
```

将 linux.yida01.com 服务器上安装好的 zookeeper 分发到 linux.yida03.com 服务器上，为了方便输入 zookeeper 命令，请在其他集群服务器上也配置好 Zookeeper 环境变量。然后修改集群中其他服务上 zookeeper 的 myid 文件，操作命令如下所示：

```
linux.yida02.com 机器上执行：
cd /opt/modules/zookeeper/zookeeper-3.4.5/tmp/data
echo 2 > myid
```

同理在 linux.yida02.com 机器上执行同样的步骤，只不过此时命令为“echo 3 > myid”，即将 myid 值改为 3，其他步骤相同。

接下来检查集群中各台服务器的防火墙是否关闭。如果未关闭，则关闭防火墙（在实际生产环境中，为了安全性是不可能关闭防火墙的，而是需要配置好哪些端口开放、哪些端口禁用等，这里只是为了方便学习）：

```
service iptables stop // 关闭 iptables 系统服务
chkconfig iptables off // 设置 iptables 不开机启动
```

然后检查集群中各台服务器的系统时间是否一致，至少要秒级同步。如果时间不同步，请先同步好系统时间。

现在你就可以分别启动集群中各服务器上的 zookeeper 服务：

```
zkServer start // 请在集群中各个节点上分别执行此命令
```

如果你能看到一个 Leader，其余都是 Follower，那表明 Zookeeper 集群配置成功了。然后在集群中任意一台服务器上连接 Zookeeper 客户端，验证数据同步性，比如先在 linux.yida02.com 服务器上创建一个 Znode：

```
zkCli.sh
create /test/ test // 创建一个 "/test" Znode
```

```
ls / // 查看所有 Znode 信息
```

然后在 linux.yida03.com 服务器上查看这个 Znode 的信息，如果能看到，这表明集群部署成功且运行正常：

```
zkCli.sh
get /test // 查询 "/test" Znode 信息
```


如果集群中服务器台数较多，登录每台服务器去手动执行 zkServer start 命令去启动 zookeeper 显然不现实，所以我们需要编写一个 shell 脚本去批量启动集群中的每台服务器。下面是我编写的一个 Zookeeper 启动脚本，仅供参考：

```
#!/bin/bash
if [ $ZOOKEEPER_HOME -eq "" ];
then
    echo "ZOOKEEPER_HOME env is not found."
    exit 1
fi
ZK_HOME=$ZOOKEEPER_HOME
ZK_CFG=$ZOOKEEPER_HOME/conf/zoo.cfg
ZK_BIN=$ZOOKEEPER_HOME/bin
if [ $# -ne 1 ];
then
    echo "Usage: $0 {start|start-foreground|stop|restart|status|upgrade|print-
cmd}" >&2
fi
if [ $ZK_HOME != "" ];
then
    alias zkServer="zkServer.sh"
    alias zk="zk.sh"
    alias zk_c="./zk.sh"

    slaves=$(cat "$ZK_CFG" | sed '/^server/!d;s/^.*=//;s/::.*$//g;/^$/d')
    for salve in $slaves ;
    do
        ssh $salve "source /etc/profile && $ZK_BIN/zkServer.sh $1"
    done
fi
```

将脚本文件放在任意路径下即可执行，执行前，请记得先赋予当前用户执行 zk.sh 脚本的权限：

```
chmod 755 zk.sh
```

 **注意** 使用批量启动 Zookeeper 脚本的前提是：你已经配置了集群中各服务器之间的免密码登录。

6. Shard Leader 选举机制

Shard Leader 的职责就是接收 Update 请求，然后将它们协调分发给 Replica。尤其是，Shard Leader 会提供以下额外的职责用于处理 Update 请求：

- 接收 Update 请求；
- 递增被 Update 的索引文档的 `_version_` 域的值，并且强制开启并发乐观锁；
- 将该索引文档写进更新日志；
- 并发地向所有 Replica 发送 Update 请求，并且阻塞直到接收到响应。

Shard Leader 在处理 Query 请求时，并没有额外的职责。任何主机上的每个 Shard 都可以是 Leader。作为一个 Replica，Leader 同样参与分布式查询请求的处理。对比 Master-Slave 架构，Master 只负责索引，Slave 只负责响应 Query 请求。而在 SolrCloud 中，所有的 Leader 和 Replica 都执行索引和查询。

在我们开始详细了解 Shard Leader 是如何选举出来之前，你应该先明白，我们不需要关心哪个节点是哪个 Shard 的 Leader。因为大多数情况下，Shard Leader 的选举细节并不会影响你搭建和操作 Solr 集群。

关于 Shard Leader 选举有两个关键点：选举初始 Leader 和当前 Leader 挂掉了会自动选举新的 Leader。Leader 选举需要集中协调，因为你不希望看到同一个 Shard 所处的两个主机都认为是 Leader。没错，ZooKeeper 在 Shard Leader 选举中扮演着重要角色。实际上，Leader 选举在许多分布式系统中是一个常见的需求。为了说明 Leader 选举是如何工作的，请考虑这样一个场景，一个 Shard 下的四个 Replica 同时并发加入集群，其中一个 Replica 想要被选举为 Leader，因此理所当然，4 个 Replica 中第一个 Replica 首先注册申请自己为候选 Leader，其他同理。在底层，Solr 使用 ZooKeeper 带序号的 Znode 来记录 Replica 节点的注册加入 ZooKeeper 的顺序。Znode 的序号是按照原子方式递增的，因此无论有多少个 Replica 节点并发地注册为候选 Leader 都没有关系。简而言之，序号值最小的 Replica 节点会在 Shard Leader 选举中获胜（先注册的序号值越小）。也就是说，如果当前的 Leader 挂掉了，在新的 Leader 选举出来之前，索引操作将无法进行，最终序号中第二小的那个正常节点会被选举为 Leader。关于 ZooKeeper 的 Leader 选举机制更加详细的内容，请读者自行查阅资料进行了解。

14.3 SolrCloud 集群搭建

下面请大家随我一起动手搭建一个 SolrCloud 集群，首先需要准备 3 台服务器，学习阶段你可以安装一个 VMware Workstation 软件，然后虚拟出 3 台 CentOS 即可。关于如何安装 VMware Workstation 以及创建 CentOS 虚拟机，这里不再赘述。在下面的示例中，我们都是 Linux 环境下进行操作，如果你对 Linux 的基本操作命令不太熟悉，请事先预习下 Linux 的基础知识。

14.3.1 在 Tomcat 容器下搭建 SolrCloud 集群

这里我们先以在 Tomcat 容器环境为例进行讲解，首先需要进行一些安装之前的准备工作，安装过程中需要使用到的系统环境如下所示：

❑ CentOS6.5 下载地址：

http://vault.centos.org/6.5/isos/x86_64/

❑ Solr6.2.1 下载地址：

<http://archive.apache.org/dist/lucene/solr/>

❑ jdk-8u111-linux-x64 下载地址：

<http://www.oracle.com/technetwork/java/javase/archive-139210.html>

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

❑ Apache Tomcat 8.5.6 下载地址：

<https://archive.apache.org/dist/tomcat>

上面每个资源文件请提前下载好，这里我们的虚拟机 CentOS 版本采用的是 6.5 版本，请尽量与本书示例保持一致，如果你在其他版本环境下进行安装操作，不能保证你 100% 能安装成功。

1. 安装 JDK

将所有需要的软件安装包都提前下载好，并存放在 /opt/software 目录下。先将 JDK 解压到 /opt/modules 目录下，命令如下所示：

```
sudo tar -xzf jdk-8u111-linux-x64.tar.gz -C /opt/modules/
```

然后配置 JDK 环境变量：

```
vi /etc/profile
JAVA_HOME=/opt/modules/jdk1.8.0_111
JRE_HOME=/opt/modules/jdk1.8.0_111/jre
PATH=$PATH:$JAVA_HOME/bin:$JRE_HOME/bin:$MAVEN_HOME/bin:$PROTOBUF_HOME/
bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$ZOOKEEPER_HOME/bin
CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar:$JRE_HOME/lib
```

 注意 最后请执行 source /etc/profile 命令使其立即生效。

2. 安装 Tomcat

首先解压 Tomcat 安装包到 /opt/modules 目录下：

```
sudo tar -xzf apache-tomcat-8.5.6.tar.gz -C /opt/modules/
```

然后配置 Tomcat 环境变量：

```
CATALINA_HOME=/opt/modules/apache-tomcat-8.5.6
PATH=$PATH:$JAVA_HOME/bin:$JRE_HOME/bin:$MAVEN_HOME/bin:$PROTOBUF_HOME/
bin:$SHADOOP_HOME/bin:$SHADOOP_HOME/sbin:$ZOOKEEPER_HOME/bin:$CATALINA_HOME/bin
```

我们需要为 Tomcat 创建系统服务，在 /etc/init.d 目录下通过 touch 命令创建一个脚本文件，名字随便取，这里暂取为 tomcat8。操作命令如下所示：

```
cd /etc/init.d
touch tomcat8
```

通过 vi 编辑我们刚刚创建的 tomcat8 脚本文件，操作命令如下所示：

```
#!/bin/bash
# description: Tomcat Start - Stop - Restart
# processname: tomcat
# chkconfig: 234 20 80
JAVA_HOME=/opt/modules/jdk1.8.0_111
export JAVA_HOME
PATH=$JAVA_HOME/bin:$PATH
export PATH
CATALINA_HOME=/opt/modules/apache-tomcat-8.5.6
case $1 in
start)
echo "Starting Tomcat"
sh $CATALINA_HOME/bin/startup.sh
;;
stop)
echo "Stopping Tomcat"
sh $CATALINA_HOME/bin/shutdown.sh
;;
restart)
echo "Restarting Tomcat"
sh $CATALINA_HOME/bin/shutdown.sh
sh $CATALINA_HOME/bin/startup.sh
;;
*)
echo $"Usage: $0 {start|stop}"
exit 1
;;
esac
exit 0
```

然后为你的 tomcat8 脚本文件添加可执行权限，命令如下：

```
sudo chmod +x tomcat8
```

将 Tomcat8 脚本文件添加进系统启动项，即作为系统服务，执行命令如下：

```
sudo chkconfig --add tomcat8
```

将 Tomcat8 服务开启并设置启动级别，执行命令如下：

```
sudo chkconfig --level 3 tomcat8 on
```

检查我们刚刚的设置是否生效, 执行如下命令:

```
sudo chkconfig --list tomcat8
```

然后执行如下命令启动 Tomcat8:

```
service tomcat8 start
```

启动成功之后, 请打开浏览器访问 <http://localhost:8080> 进行测试, 如果能看到 Tomcat 的首页界面, 则表明 Tomcat 服务启动成功了!

3. 安装单机版 Solr

先在 linux.yida01.com 机器上安装 Solr 单机版。第一步仍然是解压 Solr 安装包到 /opt/modules 目录下:

```
sudo tar -xzf /opt/softwares/solr-6.2.1.tgz -C /opt/modules/
```

然后创建 \${SOLR_HOME} 目录:

```
sudo mkdir -p /opt/solr_home
```

切换到 /opt/modules/solr-6.2.1/server/solr-webapp/webapp/WEB-INF 目录下, 编辑 web.xml 配置我们刚刚创建的 \${SOLR_HOME} 目录:

```
<env-entry>
<env-entry-name>solr/home</env-entry-name>
<env-entry-value>/opt/solr_home</env-entry-value>
<env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

在 solr-6.2.1/server/solr-webapp/webapp/WEB-INF 目录下新建 classes 目录:

```
cd /opt/modules
mkdir -p solr-6.2.1/server/solr-webapp/webapp/WEB-INF/classes
```

将 /opt/modules/solr-6.2.1/example/resources 目录下的 log4j.properties 复制到 solr-6.2.1/server/solr-webapp/webapp/WEB-INF/classes 目录下:

```
cd solr-6.2.1/server/solr-webapp/webapp/WEB-INF/classes
cp /opt/modules/solr-6.2.1/example/resources/log4j.properties ./
```

复制之前需要将 log4j.properties 修改一下, 修改内容如下所示:

```
solr.log=${solr.solr.home}/logs
```

同时在 \${SOLR_HOME} 目录下创建 logs 目录, 然后你需要将 /opt/modules/solr-6.2.1/server/lib/ext 目录下的 5 个 jar 包复制到 /opt/modules/solr-6.2.1/server/solr-webapp/webapp/WEB-INF/lib 目录下:

```
cd /opt/modules/solr-6.2.1/server/lib/ext
cp *.*.jar /opt/modules/solr-6.2.1/server/solr-webapp/webapp/WEB-INF/lib/
```

接下来需要手动打 solr.war 包:

```
cd /opt/modules/solr-6.2.1/server/solr-webapp/webapp
jar -cvf solr.war ./*
```

将 solr.war 包复制到 Tomcat 的 webapps 目录下:

```
cp solr.war /opt/modules/apache-tomcat-8.5.6/webapps/
```

然后将 /opt/modules/solr-6.2.1/server/solr 目录下的 solr.xml 和 zoo.cfg 配置文件复制到我们的 \${SOLR_HOME} 目录下:

```
cd /opt/modules/solr-6.2.1/server/solr
sudo cp solr.xml zoo.cfg /opt/solr_home/
```

启动 Tomcat 服务, 开始部署 Solr:

```
service tomcat8 start
```

最后请打开浏览器访问 <http://linux.yida01.com:8080/solr/index.html>。如果能够正常访问 Solr 的 Web 后台管理界面, 那么就说明 Solr 单机版部署成功了! 然后在其他机器上重复上面所有步骤, 当然你也可以借助 scp 命令直接远程复制。

4. SolrCloud 集群搭建

下面我们开始安装 SolrCloud 集群, 开始之前, 请确保 3 台机器之间的网络是互通的, 为了避免过多的网络传输, 建议 3 台机器在同一网段中。网络配置主要是编辑 /etc/sysconfig/network-scripts/ 下的 ifcfg-eth0 网卡配置文件, 下面是一个配置示例:

```
DEVICE=eth0
HWADDR=00:0C:29:00:37:2C
TYPE=Ethernet
UUID=5c2f07a2-0b42-4853-9fba-9993b8f5fbed
ONBOOT=yes
NM_CONTROLLED=yes
BOOTPROTO=static
IPADDR=192.168.133.100
PREFIX=24
GATEWAY=192.168.133.2
DNS1=192.168.133.2
DEFROUTE=yes
IPV4_FAILURE_FATAL=yes
IPV6INIT=no
NAME="System eth0"
```

其中 “BOOTPROTO=static” 表示采用静态 IP, IPADDR 用于配置机器的 IP 地址。紧接着请安装好 Zookeeper 集群, 具体请翻阅 14.2.11 节的内容。

修改 tomcat/bin 目录下的 catalina.sh 脚本文件，配置修改如下：

```
JAVA_OPTS="JAVA_OPTS -server -Xmx1024m -Xms256m -Dbootstrap_conf=true"
# OS specific support. $var _must_ be set to either true or false.
```

“bootstrap_conf=true”表示 Solr 会自动将 \${SOLR_HOME} 下所有 core 的配置文件上传到 Zookeeper。然后修改 /opt/solr_home 目录下 solr.xml 中的端口号：

```
# 将默认的 8983 修改为 Tomcat 的默认端口号 8080
<int name="hostPort">${jetty.port:8080}</int>
```

将修改后的 solr.xml 通过 scp 命令分发到其他两台机器上：

```
cd /opt/solr_home/
scp solr.xml linux.yida02.com:/opt/solr_home/
scp solr.xml linux.yida03.com:/opt/solr_home/
```

启动 Zookeeper 集群：

```
# 在 3 台机器上分别执行下面命令启动 Zookeeper
zkServer.sh start // 启动 Zookeeper 集群节点
```

然后分别在 3 台机器上使用如下命令检测 Zookeeper 集群状态，如果是一个 Leader，其余两个是 Follower，则 Zookeeper 集群状态正常。

```
zkServer.sh status // 查看 Zookeeper 集群中每个节点的状态
```

分别启动 3 台机器上的 Tomcat：

```
service tomcat8 start
```

我们需要创建一个 Collection，先在 /opt/solr_home 目录下创建一个“books”目录，这里“books”即 Collection 的名称。

```
mkdir books
```

在 books 目录下创建 conf 和 data 目录：

```
cd books
mkdir conf
mkdir data
```

在 conf 目录下创建 solrconfig.xml 和 schema.xml，其中 solrconfig.xml 和 schema.xml 可以从 /opt/modules/solr-6.2.1/server/solr/configsets/basic_configs/conf/ 目录下复制：

```
cd /opt/modules/solr-6.2.1/server/solr/configsets/basic_configs/conf/
cp solrconfig.xml /opt/solr_home/books/conf/
cp schema.xml /opt/solr_home/books/conf/
```

我们把 schema.xml 按如下所示修改：

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema name="example" version="1.6">
<types>
<fieldType name="string" class="solr.StrField" sortMissingLast="true" />
<fieldType name="long" class="solr.TrieLongField" precisionStep="0"
positionIncrementGap="0" />
<fieldType name="tlong" class="solr.TrieLongField" precisionStep="8"
positionIncrementGap="0" />
<fieldType name="int" class="solr.TrieIntField" precisionStep="0"
positionIncrementGap="0" />
<fieldType name="tint" class="solr.TrieIntField" precisionStep="8"
positionIncrementGap="0" />
<fieldType name="float" class="solr.TrieFloatField" precisionStep="0"
positionIncrementGap="0"/>
<fieldType name="tfloat" class="solr.TrieFloatField" precisionStep="8"
positionIncrementGap="0"/>
<fieldType name="date" class="solr.TrieDateField" omitNorms="true" />
<fieldType name="tdate" class="solr.TrieDateField" precisionStep="6"
positionIncrementGap="0"/>
</types>
<fields>
<field name="id" type="long" indexed="true" stored="true" />
<field name="book_name" type="string" indexed="true" stored="true"/>
<field name="price" type="tfloat" indexed="true" stored="true"/>
<field name="_version_" type="long" indexed="true" stored="true" />
</fields>
<uniqueKey>id</uniqueKey>
</schema>
```

这里我们定义了3个域：id、book_name 和 price，我们需要修改 solrconfig.xml 中导入 jar 包的部分配置，如下所示：

```
<lib dir="./contrib/extraction/lib" regex=".*\.jar" />
<lib dir="./dist/" regex="solr-cell-\d.*\.jar" />
<lib dir="./contrib/clustering/lib/" regex=".*\.jar" />
<lib dir="./dist/" regex="solr-clustering-\d.*\.jar" />
<lib dir="./contrib/langid/lib/" regex=".*\.jar" />
<lib dir="./dist/" regex="solr-langid-\d.*\.jar" />
<lib dir="./contrib/velocity/lib" regex=".*\.jar" />
<lib dir="./dist/" regex="solr-velocity-\d.*\.jar" />
```

然后将 solr 安装目录下的 dist 和 contrib 两个目录全部复制到当前 core 下，因为我们在 solrconfig.xml 下配置了加载这两个目录下的 jar 包，而这两个目录就是来自 solr 安装根目录下。执行如下命令：

```
cd /opt/solr_home/books
cp -r /opt/modules/solr-6.2.1/dist/ ./
cp -r /opt/modules/solr-6.2.1/contrib/ ./
```

在“books”根目录下创建一个 core.properties 文件，编辑内容如下所示：


```
cd /opt/solr_home/books/
vi core.properties
solr.shard.data.dir=/opt/solr_home/books/data/
```

将 linux.yida01.com 这台机器上的整个 SOLR_HOME 目录远程复制到其他两台机器上，执行如下命令：

```
scp -r /opt/solr_home linux.yida02.com:/opt/
scp -r /opt/solr_home linux.yida03.com:/opt/
```

最后重启 3 台机器上的 Tomcat，然后访问 <http://linux.yida01.com:8080/solr/index.html>，单击左侧的 Cloud 菜单，你将看到我们的 Solr 集群已经搭建成功！

14.3.2 在 Jetty 容器下搭建 SolrCloud 集群

在 Jetty 容器下安装 SolrCloud 集群，相对 Tomcat 来说，会简单许多，因为 Solr 安装包中内置了 Jetty 容器，其中 solr-6.2.1/server 目录其实就是一个 Jetty Server，并且 Solr 安装包中的 bin 目录下还提供很多安装脚本，用户只需要执行这些脚本文本传入适当的参数即可完成安装。下面开始详细介绍安装步骤。

首先从 solr 的安装包中抽离出安装脚本文件 install_solr_service.sh，执行如下命令：

```
sudo tar -xvzf solr-6.2.1.tgz solr-6.2.1/bin/install_solr_service.sh --strip-components=2
```

然后执行 ./install_solr_service.sh-help 命令可以查看到该脚本的执行语法，下面对 install_solr_service.sh 脚本文件支持的参数进行详细说明：

install_solr_service.sh 脚本文件的第一个参数必须是 Solr 的安装包，比如“solr-5.0.0.tgz”，安装包文件仅仅支持 .tgz 或者 zip 压缩格式。第一个参数为必需参数，install_solr_service.sh 脚本还支持以下可选参数：

- ❑ -d：用于配置 Solr 的日志、pid 文件，或者索引数据等文件的目录，即脚本中定义的 \$SOLR_VAR_DIR 变量，如果未设置，默认值为 /var/solr。
- ❑ -i：用于配置 Solr 安装包文件解压后的安装目录，默认是安装到 /opt 目录下，如果你需要设置为自定义目录，那么该指定目录必须存在。
- ❑ -p：用于配置 Solr 服务绑定的端口号，其实就是配置 Solr 内置 Jetty 容器的启动端口号，默认值是 8983，即 Jetty 容器的默认启动端口号。
- ❑ -s：用于配置 Solr 服务的名称，即你访问 solr 时 <http://ip:8983/solr>，这里的 solr 即 Solr 服务名称，默认值是 solr。
- ❑ -u：用于配置 Solr 安装包解压后得到的所有文件的所有者，以及运行 Solr 服务使用的系统用户名，默认使用的是 Solr 用户，如果你指定的系统用户不存在，install_solr_service.sh 脚本会自动创建该用户。
- ❑ -f：用于配置 Solr 升级，它会重写之前安装生成的初始化脚本以及连接。



注意 install_solr_service.sh 脚本文件必须使用 root 用户进行执行。

首先请创建 \$SOLR_VAR_DIR 目录，这里我们创建 /opt/solr6 目录作为 \$SOLR_VAR_DIR 目录，然后在 /opt/solr6 目录下创建 logs 目录作为 solr 的日志目录，在 /opt/solr6 目录下创建 pid 目录作为 solr 的 pid 进程文件的存放目录，执行命令如下所示：

```
mkdir -p /opt/solr6
cd /opt/solr6
mkdir logs
mkdir pid
```

创建 /opt/solr_home 目录作为我们的 \${SOLR_HOME} 目录，执行命令如下所示：

```
mkdir -p /opt/solr_home
```

将 /opt/solr-6.2.1/example/example-DIH/solr 目录下 solr.xml 复制到 /opt/solr_home 目录下，同时对 solr.xml 稍作配置，示例如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<solr>
  <str name="shareSchema">${shareSchema:false}</str>
  <solrcloud>
    <str name="host">linux.yida01.com</str>
    <int name="hostPort">${hostPort:8983}</int>
    <str name="hostContext">${hostContext:solr}</str>
    <bool name="genericCoreNodeNames">${genericCoreNodeNames:true}</bool>
    <int name="leaderVoteWait">3000</int>
    <int name="distribUpdateSoTimeout">${distribUpdateSoTimeout:120000}</int>
    <int name="distribUpdateConnTimeout">${distribUpdateConnTimeout:15000}</int>
    <str name="zkHost">linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181</str>
    <int name="zkClientTimeout">${solr.zkclienttimeout:30000}</int>
  </solrcloud>
  <shardHandlerFactory name="shardHandlerFactory" class="HttpShardHandlerFactory">
    <str name="urlScheme">${urlScheme:}</str>
    <int name="socketTimeout">${socketTimeout:120000}</int>
    <int name="connTimeout">${connTimeout:15000}</int>
  </shardHandlerFactory>
</solr>
```

重点是其中的 host 和 zkHost 两个配置项，host 表示当前 solr 节点的 ip 或域名，当你稍后将 solr.xml 分发到其他 Solr 节点时，请记得修改 host 配置项。此外配置此文件时，切记正确使用 <str> 和 <int> 元素，比如 zkHost 配置项应该是一个 Zookeeper 集群连接字符串，即你应该使用 <str> 元素，如果你粗心使用 <int> 元素进行配置，就会导致 Solr 初始化失败。

请从 solr-6.2.1.tgz 安装包中抽离出 solr.in.sh 脚本文件，执行如下命令：

```
cd /opt/software
sudo tar -xvzf solr-6.2.1.tgz solr-6.2.1/bin/solr.in.sh --strip-components=2
```

将抽离出来的 `solr.in.sh` 复制到 `/opt/solr6` 目录下：

```
cp solr.in.sh /opt/solr6
```

通过 `vi` 对 `solr.in.sh` 进行编辑，编辑内容如下所示：

```
SOLR_JAVA_HOME="/opt/modules/jdk1.8.0_111"
SOLR_JAVA_MEM="-Xms512m -Xmx1024m"
SOLR_HOST="linux.yida01.com"
SOLR_TIMEZONE="Asia/Shanghai"
ENABLE_REMOTE_JMX_OPTS="false"
SOLR_OPTS="$SOLR_OPTS -Dsoler.clustering.enabled=true-Dbootstrap_conf=true"
SOLR_HOME="/opt/solr_home"
LOG4J_PROPS="/opt/solr6/log4j.properties"
SOLR_LOGS_DIR="/opt/solr6/logs"
SOLR_PORT="8983"
SOLR_PID_DIR="/opt/solr6/pid"
ZK_HOST="linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181"
ZK_CLIENT_TIMEOUT="30000"
SOLR_HEAP="512m"
SOLR_OPTS="$SOLR_OPTS -Xss256k"
RMI_PORT=18983
```

- **SOLR_JAVA_HOME**：用于配置 Solr 需要用到的 `JAVA_HOME`，如果你已经在系统环境变量中配置了 `JAVA_HOME`，就可以不需要配置 `SOLR_JAVA_HOME`，如果不配置 `SOLR_JAVA_HOME`，Solr 会默认查找系统环境变量中的 `JAVA_HOME`。
- **SOLR_JAVA_MEM**：用于配置 JVM 内存。
- **SOLR_HOST**：用于配置当前 Solr 节点的主机 ip 或者域名，默认值为 `localhost`。
- **SOLR_TIMEZONE**：用于配置 Solr 的使用时区，默认使用的是“UTC”时区。
- **ENABLE_REMOTE_JMX_OPTS**：用于配置是否开启 JMX 远程访问，默认值是 `false`。
- **SOLR_OPTS**：在 `SOLR_OPTS` 里添加的参数最终都会追加到 Java 启动命令参数中，前提是你添加了 `-a` 选项。
- **SOLR_HOME**：用于配置 Solr 的 `${SOLR_HOME}` 目录。
- **LOG4J_PROPS**：用于配置自定义的 `log4j` 配置文件，默认值为 `/var/solr`。
- **SOLR_LOGS_DIR**：用于配置 Solr 服务的日志文件存放目录，默认值为 `/var/solr/logs`。
- **SOLR_PORT**：用于配置 Solr 服务监听的端口号，其实就是 Jetty 容器的启动端口号，默认值为 `8983`。
- **SOLR_PID_DIR**：用于配置 Solr 服务的进程 PID 文件的存放目录，默认值为 `/var/solr`。
- **ZK_HOST**：用于配置 SolrCloud 需要连接的 Zookeeper 集群节点的 ip 和端口，多个采用逗号分隔。
- **ZK_CLIENT_TIMEOUT**：用于配置连接 Zookeeper 的最大超时时间，单位是毫秒。

□ SOLR_HEAP: 用于配置 Java 堆内存。

□ SOLR_OPTS="\$SOLR_OPTS-Xss256k": 用于配置配置 Java 栈空间大小。

□ RMI_PORT: 用于配置 Solr 的 RMI 端口号, 默认值为 SOLR_PORT + 10000。

执行解压出来的 install_solr_service.sh 脚本文件:

```
sh install_solr_service.sh solr-6.2.1.tgz -i /opt -d /opt/solr6 -u solr -s
solr -p 8983
```

需要注意的是, 在执行安装脚本之前, 请确保系统的 solr 用户拥有对 /opt/solr_home 目录的写权限。编辑 /opt/solr6 目录下的 log4j.properties (此配置文件可以从 /opt/solr-6.2.1/example/resources/ 目录下复制) 配置文件, 将其中:

```
solr.log=${solr.solr.home}/../logs
```

修改为:

```
solr.log=/opt/solr6/logs
```

同理, 在其他两台机器上重复上述步骤, 即可完成 Solr 安装。这里为了简便, 我们可以将 /opt/solr_home 和 /opt/solr6 这两个目录直接整个远程复制到另外两台机器上:

```
scp -r /opt/solr_home linux.yida02.com:/opt/
scp -r /opt/solr_home linux.yida03.com:/opt/
scp -r /opt/solr6 linux.yida02.com:/opt/
scp -r /opt/solr6 linux.yida03.com:/opt/
```

分别在另外两台机器上创建 solr 用户:

```
useradd solr
passwd solr
```

设置 /opt/solr6 和 /opt/solr_home 两个目录的所有者都是 solr 用户:

```
cd /opt
sudo chown -R solr:solr solr6
sudo chown -R solr:solr solr_home
```

将抽离出来的 install_solr_service.sh 脚本也远程复制到另外两台机器的 /opt/software 目录下:

```
cd /opt/software
scp install_solr_service.sh linux.yida02.com:/opt/software
scp install_solr_service.sh linux.yida03.com:/opt/software
```

将 solr.in.sh 脚本文件也远程复制到另外两台机器的 /opt/solr6 目录下:

```
scp /etc/default/solr.in.sh linux.yida02.com:/opt/solr6
scp /etc/default/solr.in.sh linux.yida03.com:/opt/solr6
```

solr.in.sh 脚本文件远程复制完成之后, 请务必记得修改其中的 SOLR_HOST="linux.

yida01.com", 因为每台机器的 IP 或域名不相同, 否则在启动 Solr 服务时会抛出如下异常:

```
org.apache.solr.common.SolrException: A previous ephemeral live node still
exists. Solr cannot continue. Please ensure that no other Solr process using the
same port is running already.
atorg.apache.solr.cloud.ZkController.checkForExistingEphemeralNode
(ZkController.java:741)
```

然后在另外两台机器的 /opt/softwares 目录下, 执行如下命令安装 Solr (注意, 必须使用 root 用户执行):

```
sh install_solr_service.sh solr-6.2.1.tgz -i /opt -d /opt/solr6 -u solr -s
solr -p 8983
```

在 3 台服务器上全部安装成功之后, 你最好能够额外设置 /opt/ 几个目录的所有者为 solr 用户, 因为我们以后将以 solr 用户来操作, 设置其所有者为 solr 用户可以避免操作权限问题:

```
cd /opt/
chown -R solr:solr solr-6.2.1
chown -R solr:solr solr
```

通过如下命令来查看当前 Solr 集群节点的状态信息:

```
service solr status // 查看某个 Solr 节点的状态
```

返回的节点状态信息大致如下:

```
{
  "solr_home":"/opt/solr_home",
  "version":"6.2.1 43ab70147eb494324a1410f7a9f16a896a59bc6f - shalin - 2016-09-
15 05:20:53",
  "startTime":"2016-11-02T01:13:53.413Z",
  "uptime":"0 days, 0 hours, 2 minutes, 50 seconds",
  "memory":"37.2 MB (%7.6) of 490.7 MB",
  "cloud":{
    "ZooKeeper":"linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.
com:2181",
    "liveNodes":"3",
    "collections":"1"}}}
```

执行如下命令来管理当前 Solr 节点:

```
service solr start // 启动当前 Solr 节点
service solr restart // 重新启动当前 Solr 节点
service solr stop // 关闭当前 Solr 节点
```

如果你发现执行 service solr start 命令提示 “env:/etc/init.d/solr: Permission denied”, 即你没有权限执行该脚本, 那么此时需要将该脚本文件的所有者设置为当前 solr 用户, 操作命令如下:

```
chown -R solr:solr /etc/init.d/solr
```

最后请打开浏览器访问 <http://linux.yida01.com:8983/solr/index.html>，单击左侧的 Cloud 菜单，你会看到我们的 SolrCloud 状态示意图。至此，SolrCloud 在 Jetty 容器下搭建也完成了。

如果你在安装过程中操作失误导致 SolrCloud 安装失败，请执行如下脚本命令进行撤销：

```
service solr stop
zkServer stop
rm -rf /opt/solr6/pid/*
rm -rf /opt/solr
rm -rf /opt/solr-6.2.1
cp /etc/default/solr.in.sh /opt/solr6
rm -rf /etc/default/solr.in.sh
rm -rf /etc/init.d/solr
rm -rf /opt/solr6/logs/solr*.log
rm -rf /tmp/*
rm -rf /opt/modules/zookeeper/zookeeper-3.4.6/tmp/data/zookeeper_server.pid
```

然后重新启动 Zookeeper，启动 Zookeeper 之前先通过 `lsof-i:2181` 命令检查 Zookeeper 的 2181 端口是否已经被占用，如果被占用了，请使用 `kill-9` 命令杀掉该进程，再启动 Zookeeper。切到 `/opt/softwares` 目录下重新执行 `install_solr_service.sh` 安装脚本。安装过程中如果出现任何异常，请仔细查看 `/opt/solr6/logs/` 目录下的日志文件。

14.4 SolrCloud 的基本操作

目前我们已经知道如何搭建 SolrCloud 集群，下面让我们开始学习一些 SolrCloud 的基本操作。为了简化用户的集群操作，Solr 分别在 `/opt/solr-6.2.1/bin` 和 `/opt/solr-6.2.1/server/scripts/cloud-scripts/` 目录下提供了很多很实用的脚本文件，用于实现一些常用的 Solr 操作。由于脚本文件目录层级较深，为了避免每次必须切换到脚本文件所处目录下才能执行脚本命令的窘境，这里我建议大家为 Solr 设置一下系统环境变量，这样就能在任意路径下执行这些脚本命令了。

14.4.1 Solr 环境变量设置

使用 root 用户执行 `vi/etc/profile` 命令，然后编辑如下内容：

```
#Solr
SOLR_BASE=/opt/solr-6.2.1/
SOLR_ZK=/opt/solr-6.2.1/server/scripts/cloud-scripts
PATH=$PATH:$JAVA_HOME/bin:$JRE_HOME/bin:$MAVEN_HOME/bin:$PROTOBUF_HOME/
bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$ZOOKEEPER_HOME/bin:$CATALINA_HOME/
bin:$SOLR_BASE/bin:$SOLR_ZK
```

编辑完成之后，使用 `source/etc/profile` 命令使上述修改立即生效。然后在其他机器重复

上面操作。最后我们可以使用 `echo $SOLR_ZK` 命令来验证我们的 SOLR 环境变量配置是否生效，如果能够正常打印出我们配置的脚本路径，那么说明环境变量配置已经生效。

14.4.2 创建 Collection

执行 `solr create_collection-help` 命令，我们可以看到使用 `solr` 脚本命令创建一个 Collection 的基本语法：

```
solr create_collection [-c collection] [-d confdir] [-n configName] [-shards #]
[-replicationFactor #] [-p port]
```

- ❑ `-c<collection>`：表示你需要创建的 Collection 名称。
- ❑ `-d<confdir>`：表示 Collection 需要的配置文件目录，目录下主要就是 `solrconfig.xml` 和 `schema.xml`。内置可选值有 `basic_configs`、`data_driven_schema_configs`、`sample_techproducts_configs`。其实这 3 个可选项就对应着 `solr-6.2.1/server/solr/configsets` 目录下的 3 个配置示例。当然，你还可以自定义配置文件目录，比如：`solr create_collection -c mycoll -d tmp/myconfig`。但是前提是，你自定义的目录已经存在且该目录下存在 `solrconfig.xml` 和 `schema.xml`。如果此参数未设置，默认值为 `data_driven_schema_configs`。
- ❑ `-n<configName>`：表示配置文件目录在 Zookeeper 中的名称，默认 `-d` 参数下的所有配置文件会上传到 Zookeeper 中且以 Collection 名称（即 `-c` 参数）作为配置文件目录在 Zookeeper 中的名称，如果想要自定义，那么请指定此参数。
- ❑ `-shards`：用于指定当前 Collection 分成几个 Shard，默认值为 1。
- ❑ `-replicationFactor`：用于指定当前 Collection 的每个 Shard 划分几个 Replica，默认值为 1。
- ❑ `-p <port>`：即你想要使用本地哪个 Solr 实例来创建这个 Collection，如果没有指定此参数，那么脚本就会在本地系统中查询正在运行的 Solr 实例，使用找到的第一个 Solr 实例绑定的端口号。因为本地可能存在多个 Solr 实例，而本地多个 Solr 实例区分的标志就是 port 端口号。

根据上面的语法得知，如果想要自定义 Collection 的配置文件，需要提前准备好 Collection 的 `solrconfig.xml` 和 `schema.xml`，这里可以直接从我们之前创建的“books”Collection 复制即可，或者直接使用 Solr 内置提供的 3 个配置文件示例。具体操作请执行如下命令：

```
cd /opt/solr_home/           // 先切换到 SOLR_HOME 目录下
mkdir users                  // 创建 collection 的目录
chown -R solr:solr users     // 将新创建的 users 目录的所有者设置为 solr 用户
// 将 books collection 中的配置文件和相关 jar 包复制到 collection 中
cp -r books/conf books/dist books/contrib ./users
```

然后你可以按照实际项目需求对 `users/conf` 目录下 `schema.xml` 中的域和域类型定义进行适当修改。Collection 的配置文件准备好了之后，`create_collection` 命令的 `-d` 参数值就可以指定为：`/opt/solr_home/users/conf`。然后我们就可以执行如下命令来创建一个新的名称为

“users”的 Collection:

```
solr create_collection -c users -d /opt/solr_home/users/conf -shards 3
-replicationFactor 4
```

执行该命令过程中,你会发现实际脚本最终还是通过访问如下链接来实现 Collection 创建,其实操作两者效果是等价的。当然你也可以借助 SolrJ 来调用上面的请求 URL 以实现 Collection 的创建。不管是使用 SolrJ 还是使用 Solr 提供的脚本文件,最终都是通过 Solr 提供的 Collection REST API 来实现 Collection 的创建:

```
http://localhost:8983/solr/admin/collections?action=CREATE&name=users&numShards
=3&replicationFactor=4&maxShardsPerNode=4&collection.configName=users
```

其中,“action=CREATE”表示执行 Collection 创建操作,同理还有“action=DELETE”表示删除一个 Collection; name 表示 collection 的名称; numShards 表示需要对 Collection 划分几个 Shard; replicationFactor 参数表示每个 Shard 划分几个 Replica; collection.configName 参数表示我们的“users”Collection 的配置文件目录上传到 Zookeeper 使用的名称; maxShardsPerNode 参数表示每个 Solr 节点上最多允许有几个 Shard。

14.4.3 删除 Collection

删除一个 Collection 的脚本命令语法如下所示:

```
solr delete [-c name] [-deleteConfig true|false] [-p port]
```

❑ -c <name>: 表示想要删除的 Collection 或者 Core 的名称, Solr 会自动判断当前 Solr 节点是运行在单机模式下还是 SolrCloud 模式下。如果是 SolrCloud 模式下,那么就是删除 Collection。

❑ -deleteConfig: boolean 参数,表示删除 Collection 的同时是否从 Zookeeper 删除 Collection 相关的配置文件,默认值为 true。

❑ -p <port>: 表示想要用来删除 Core 或 Collection 本地运行中的 Solr 实例对应的端口号。如果此参数未指定,那么会使用本地查找到的第一个 Solr 实例的端口号。

删除 Collection 对应 REST API 接口的命令如下:

```
http://linux.yida02.com:8983/solr/admin/collections?action=DELETE&name=users
```

其中, /admin/collections 为 Collection 删除操作对应的请求 Path; action=DELETE 表示当前执行的 Collection 的删除操作; name 参数表示想要删除的 Collection 名称。

14.4.4 启动 Solr

启动 Solr 的脚本命令语法如下所示:

```
solr start [-f] [-c] [-h hostname] [-p port] [-d directory] [-z zkHost] [-m
```

```
memory] [-e example] [-s solr.solr.home] [-a "additional-options"] [-V]
```

- ❑ **-f**: 表示在前台运行 Solr 服务。
- ❑ **-c 或 -cloud**: 表示以 SolrCloud 模式启动 Solr 服务, 如果同时 **-z** 参数未指定, 那么此时会使用内置的 Zookeeper, Zookeeper 的默认启动端口号为 Solr 的端口号 + 1000, 比如 Solr 端口号为 8983, 那么 Zookeeper 的启动端口号就是 9983。
- ❑ **-h <host>**: 指定 Solr 实例的主机 ip 或域名。
- ❑ **-p <port>**: 用于指定 Solr 实例的启动端口号, 默认值是 8983。Solr 默认停止端口号等于 Solr 启动端口号减去 1000。JMX RMI 监听端口号默认等于“1”+ Solr 实例的启动端口号, 假如 Solr 实例的启动端口号为 8985, 那么 Solr 实例的停止端口号为 7985, JMX RMI 监听端口号为 18985。
- ❑ **-d <dir>**: 指定 Solr 服务运行 Web 容器的根目录, 即 Jetty 容器的根目录。当你想要使用自己外部安装的 Jetty 容器而不是使用 Solr 内置的 Jetty 容器时, 你可能需要指定此参数, 默认值为 server。
- ❑ **-z <zkHost>**: Zookeeper 的连接字符串, 只有运行在 SolrCloud 模式下才需要设置此参数, 多个 Zookeeper 节点的 ip 和端口号请使用逗号分隔, 比如 ip1:2181,ip2:2181,ip3:2181。当你使用内嵌的 Zookeeper 时, 不需要指定此参数。
- ❑ **-m <memory>**: 设置 JVM 的 **-Xms** 和 **-Xmx** 参数, 比如 **-m 4g** 即表示 **-Xms4g-Xmx4g**, 默认脚本会设置堆内存为 512M。
- ❑ **-s <dir>**: 用于设置 SOLR_HOME 目录, 即 solr.solr.home 系统参数, Solr 会在该目录下创建 Core。SOLR_HOME 目录下应该包含一个 solr.xml, 除非 solr.xml 已经上传到 Zookeeper。当指定了 **-e** 参数时, 此参数会被忽略。默认值为 server/solr。
- ❑ **-e <example>**: 表示启动哪个 Solr 示例, 可选的 Solr 示例有:
 - cloud: SolrCloud 的示例。
 - techproducts: 演示很多 Solr 核心功能的综合示例。
 - dih: Solr DIH 数据导入示例。
 - schemaless: 无 Schema 模式示例。
- ❑ **-a**: 用于 Solr 启动时传递给 JVM 的额外参数, 比如 **-a "-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=18983"** 这些额外的参数需要使用双引号包裹起来。
- ❑ **-noprompt**: 当运行 Solr 示例时, 执行命令过程中不提示进行任何输入。
- ❑ **-V**: 显式脚本执行过程的详细信息。

与 solr start 命令对应, 同理还有 solr restart 命令, 用于实现 Solr 节点重启, 可以使用的参数全部相同, 这里不再赘述。

14.4.5 停止 Solr

停止 Solr 的脚本命令语法如下所示:

```
solr stop [-k key] [-p port] [-V]"
```

□ -k：用于停止 Solr 的 key，其实这是 Jetty 的 stop key，设计 stop key 是为了安全考虑，即你只有知道了 stop key 才能停止 Jetty 服务。

□ -p：停止 solr 的端口号，默认值为 Solr 启动端口号减去 1000。

□ -V：表示开启打印命令执行过程中的详细信息，一般用于调试。

14.4.6 查看 Solr 状态

查看 Solr 状态的命令是 `solr status`，此命令没有任何可选的输入参数，它会返回当前机器上运行中的所有 Solr 实例的状态信息。

14.4.7 Collection 健康检测

Solr 的 Collection 健康检测的脚本命令语法如下所示：

```
solr healthcheck [-c collection] [-z zkHost]
```

其中，-c 参数表示你需要对哪个 Collection 执行健康检测，-z 参数表示 SolrCloud 集群所使用的 Zookeeper 集群所有节点的 ip 和端口号。下面是一个对“goods”这个 Collection 执行健康检测的简单示例：

```
solr healthcheck -c goods -z linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181
```

返回的状态大致如下：

```
{
  "collection": "goods",
  "status": "healthy",
  "numDocs": 0,
  "numShards": 4,
  "shards": [
    {
      "shard": "shard1",
      "status": "healthy",
      "replicas": [
        {
          "name": "core_node9",
          "url": "http://linux.yida03.com:8983/solr/goods_shard1_replica4/",
          "numDocs": 0,
          "status": "active",
          "uptime": "0 days, 5 hours, 18 minutes, 45 seconds",
          "memory": "67.6 MB (%13.8) of 490.7 MB",
          "leader": true,

```

从返回的状态信息可以看出，`solr healthcheck` 命令会返回指定 Collection 的详细信息，包括当前 Collection 的状态，`healthy` 即表示健康良好。`numDocs` 表示当前 Collection

下包含多少个 Document。numShards 表示当前 Collection 被划分成几个 Shard。shards 下表示的是当前 Collection 下包含的所有 Shard 的详细信息。replicas 表示每个 Shard 下包含的 Replica 详细信息，其中：name 表示当前 Replica 的名称，url 表示当前 Replica 的 Core URL，numDocs 表示当前 Replica 下包含了多少个 Document，uptime 表示当前 Replica 上一次更新时间，memory 表示当前 Replica 的内存占用情况，leader 表示当前 Replica 是否为所属 Shard 的 Leader。

14.4.8 管理 Zookeeper 上的配置文件


通过 solr 脚本管理 Zookeeper 上配置文件的基本命令语法如下所示：

```
solr zk upconfig|downconfig -d <confdir> -n <configName> [-z zkHost]
solr zk cp [-r] <src><dest> [-z zkHost]
solr zk rm [-r] <path> [-z zkHost]
solr zk mv <src><dest> [-z zkHost]
solr zk ls [-r] <path> [-z zkHost]
```

solr zk upconfig 命令用于从本地机器上传配置文件到 Zookeeper 中，同时兼容旧版本中的 solr zk upconfig。-d <confdir> 参数表示一个本地文件系统上你想要上传到 Zookeeper 的配置文件目录，如果你配置的是相对路径，那么默认是相对“solr/server/solr/configsets”，否则会当作本地文件系统里一个普通的目录绝对路径。-n <configName> 参数表示配置目录在 Zookeeper 上的名称。-z 参数用于指定 Zookeeper 的访问地址，如果有多个，则用逗号分隔。当你修改某个 Collection 的 solrconfig.xml 或 schema.xml 时，需要先本地修改好，然后通过此命令上传到 Zookeeper 上，最后重新加载你的 Collection 使修改立即生效。

同理，solr zk downconfig 用于从 Zookeeper 下载配置文件到本地机器，同时向后兼容 solr zk-downconfig。

solr zk cp 命令用于本地文件（或文件夹）与 Zookeeper 文件（或文件夹）之间的复制，也有可能是 Zookeeper 上的文件（或文件夹）复制。<src>、<dest> 既可以表示本地文件也可以表示 Zookeeper 上的文件，若表示 Zookeeper 上的文件，则必须使用 zk:/ 前缀。

 **注意** 是 zk:/ 不是 zk:，因为 Zookeeper 上的所有路径都必须是绝对路径，开头的斜杠不能省略。如果你需要表示本地的文件路径，那么最好加上 file:，其中 file:/ 开头表示本地文件系统里的绝对路径，而 file: 表示本地文件系统里的相对路径。<src>、<dest> 的路径表达式都不支持通配符。

solr zk rm 用于删除 Zookeeper 上的文件（或文件夹），-r 参数表示是否递归删除一个目录，如果 <path> 是一个目录，但是 -r 参数未指定，那么命令会执行失败。<path> 表示一个 Zookeeper 上文件或文件夹的路径，路径表达式为 [zk:]/path/to/zk/node，其中 zk: 前缀可以省略。抛开前缀而言，路径必须以斜杠开头，比如 path/to/zk/node 是错误的，而 /path/to/zk/

node 是正确的。<path> 参数值不能是 /，即你 cannot 通过此命令删除 Zookeeper 的根 Znode。

`zk mv <src><dest> [-z zkHost]` 命令用于剪切或重命名 Zookeeper 上的 Znode。<src>、<dest> 表示 Zookeeper 上的节点路径表达式，指定格式为 `zk:/path/to/zk/node`，其中 `zk`：前缀可以省略。

`solr zk ls [-r] <path> [-z zkHost]` 命令用于列出 Zookeeper 上指定节点 `path` 下的 Znode 名称信息。`-r` 参数表示是否递归列举所有 Znode 名称信息。<path> 表示 Zookeeper 上的节点 `path` 表达式。注意，此命令只会列出 Znode 的节点名称，并不会显示 Znode 上包含的数据。

除此之外，Solr 的 `server/scripts/cloud-scripts` 下提供了 `zkcli.sh` 脚本。通过此脚本你可以启动 Solr、上传下载 Solr 配置文件、创建一个 Zookeeper 的 Znode 节点、删除一个 Zookeeper 的 Znode 节点、设置某个 Znode 节点的值等。`zkcli.sh` 脚本的基本使用语法是：

```
zkcli.sh -cmd command -z zkHost
```

其中 `-cmd` 参数的可选值有：`bootstrap`, `upconfig`, `downconfig`, `linkconfig`, `makepath`, `put`, `putfile`, `get`, `getfile`。由于 `zkcli.sh` 脚本都是对 Zookeeper 服务上的节点进行操作，因此你必须添加 `-z` 参数指定 Zookeeper 访问地址。

```
zkcli.sh -cmd bootstrap -s /opt/solr_home -z linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181
```

上面命令表示启动一个 Solr 节点，其中 `-s` 参数表示当前 Solr 节点对应的 `SOLR_HOME` 目录。当然你也可以使用前面我们讲过的 Solr 脚本来启动。

```
zkcli.sh -cmd upconfig -d /opt/solr_home6/configsets/user/conf/ -n user -c user -z linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181
```

上面的脚本命令表示将本地的 `/opt/solr_home6/configsets/user/conf/` 目录下所有配置文件上传到 Zookeeper 上，其中 `-n` 表示上传到 Zookeeper 后它在 Zookeeper 上的配置文件目录名称。`-c` 参数表示当前配置文件属于哪个 Collection。

```
zkcli.sh -cmd downconfig -d /opt/solr_home6/configsets/user/ -n user -z linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181
```

这条脚本命令表示将 Zookeeper 上名称为 “user” 的配置文件目录整个下载到本地的 `/opt/solr_home6/configsets/user/` 目录下。

`upconfig` 和 `downconfig` 是用来批量上传和下载整个配置文件目录的，如果你只想上传单个配置文件时，需要使用 `zkcli.sh-cmd putfile` 命令，下面是一个简单使用示例：

```
zkcli.sh -z linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181 -cmd putfile /configs/joinTest/schema.xml /opt/solr6/configsets/joinTest/conf/schema.xml
```

上面的脚本命令表示将本地 `/opt/solr6/configsets/joinTest/conf` 目录下的 `schema.xml` 配置文件上传到 Zookeeper 的 `/configs/joinTest/schema.xml` 路径下。同理还有 `zkcli.sh-cmd`

getfile 命令，它用于从 Zookeeper 上下载单个配置文件到本地目录，用法与 putfile 几乎一样。当你需要单个修改集群中某个 Collection 中的配置文件时，可能需要先使用 getfile 下载配置文件到本地，本地修改完成之后再使用 putfile 上传到 Zookeeper。

如果你在使用 zkcli.sh-cmd upconfig 命令上传配置文件目录到 Zookeeper 时忘记添加 -c 参数，即你的配置文件暂时没有跟任何 Collection 进行关联，那么你的 Collection 将无法应用这些配置文件，此时你可以使用 zkcli.sh-cmd linkconfig 命令显式地将一个配置文件目录与指定的 Collection 进行绑定。示例如下：

```
zkcli.sh -cmd linkconfig -n user -c user -z linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181
```

同理还有 zkcli.sh-cmd get，与 Zookeeper 客户端命令中的 get 命令用法类似，zkcli.sh-cmd put 与 Zookeeper 客户端命令中的 set 命令用法类似，zkcli.sh-cmd clear 与 Zookeeper 客户端命令中的 delete 命令用法类似，zkcli.sh-cmd list 与 Zookeeper 客户端命令中的 ls 命令用法类似。由于这几个命令与 Zookeeper 客户端命令用法几乎一模一样，因此这里就不详细展开了。

14.5 SolrCloud 配置详解

与 SolrCloud 紧密相关的两个配置文件就是 `${SOLR_HOME}` 目录下 `solr.xml` 和 `zoo.cfg` 两个配置。`solr.xml` 必须处于 `${SOLR_HOME}` 根目录下，除非你已经将 `solr.xml` 上传到 Zookeeper 上。`zoo.cfg` 其实就是 Zookeeper 的配置文件，对 Zookeeper 熟悉的同学来说，应该不会陌生。

14.5.1 solr.xml 详解

在早期的 Solr 版本中，Solr Core 是在 `solr.xml` 中通过 `<core>` 元素定义的，随着 Solr 版本不断升级，`solr.xml` 已经慢慢进化成 SolrCloud 的配置“阵地”。之前的 Core 配置已经从 `solr.xml` 转移到 `core.properties` 属性文件中。从 Solr 的 4.3 版本开始，如果一个 Core 目录下包含了 `core.properties` 属性文件，那么 Solr 会自动加载该 Core，即 Core 的自动发现机制。你不需要在 `solr.xml` 中显式的定义 Core 了。旧版本中的 Core 定义是这样的：

```
<solr persistent="true">
  <cores adminPath="/admin/cores" defaultCoreName="collection1" host="${host:}"
    hostPort="${jetty.port:}" hostContext="${hostContext:}"
    zkClientTimeout="${zkClientTimeout:15000}">
    <core name="collection1" instanceDir="collection1" />
    <core name="collection2" instanceDir="collection2" />
  </cores>
</solr>
```

自从 Solr 4.3 版本开始，Solr 建议你使用 `core.properties` 来定义一个 Core，`core.properties`

配置示例如下：

```
name=clothes
config=solrconfig.xml
schema=schema.xml
dataDir=data
```

下面详细列举了 core.properties 中支持的配置属性，如表 14-2 所示。

表 14-2 core.properties 配置属性表

属 性	描 述
name	SolrCore 的名称，跟 Core 目录名称无关
config	solrconfig.xml 文件名称，默认值就是 solrconfig.xml
schema	schema.xml 文件名称，默认值就是 schema.xml
dataDir	Core 的数据目录，用于存放索引数据和事务日志
configSet	共享配置目录名称，具体请翻到 13.9.1 章节进行了解
properties	core.properties 属性文件的路径，可以是一个绝对路径，也可以是一个相对 Core 的 instanceDir 目录的相对路径
loadOnStartup	是否在 Solr 启动时自动加载
uLogDir	指定当前 Core 的事务日志存放目录，可以是绝对路径，也可以是相对 Core 的 instanceDir 目录的相对路径
collection	指定这个 Core 属于哪个 Collection，用于 SolrCloud 模式下
shard	指定当前 Core 属于哪个 Shard，用于 SolrCloud 模式下
collection.configName	指定当前 Core 的配置文件在 Zookeeper 上的名称，用于 SolrCloud 模式下
transient	如果此属性设置为 true，那么当 Solr 达到 transientCacheSize 给出的限定值时，该 core 可能会被卸载（unload），默认值是 false。当 transient=true 时，最近最少被使用的 core 会优先被卸载

你还可以在 core.properties 属性文件中自定义任何的属性，比如设置 key=value，然后你就可以在 solrconfig.xml 中以 "\${key : 默认值}" 的方式来引用 core.properties 中自定义的属性。正由于 Core 定义已经由 solr.xml 集中定义转移到每个 Core 目录下的 core.properties 属性文件中单独定义，使得 solr.xml 慢慢变成专门为 SolrCloud 配置所服务的。从 Solr5.0 开始，solr.xml 的配置如下所示：

```
<solr>
<solrcloud>
<str name="host">${host:}</str>
...// 其他省略
</solrcloud>
<shardHandlerFactory name="shardHandlerFactory"
  class="HttpShardHandlerFactory">
<int name="socketTimeout">${socketTimeout:600000}</int>
<int name="connTimeout">${connTimeout:60000}</int>
</shardHandlerFactory>
</solr>
```

其中 <solr> 元素是 solr.xml 的根元素，它支持如表 14-3 所示的几个可以配置的属性。

表 14-3 <solr> 参数表

属 性	描 述
coreLoadThreads	用于指定加载 Core 时所使用的线程数量
persistent	用于指定通过 Solr API 或者通过 Solr 管理后台界面对 solr.xml 造成的修改是否应该持久化到 solr.xml 文件中，默认值为 true。如果 persistent=true，你需要确保当前 Solr 服务的运行用户拥有对 solr.xml 文件的写权限，否则 Solr 会抛出 IO 异常
sharedLib	用于指定所有 core 的共享 lib 目录，详细请翻到 2.5.3 章节
zkHost	在 SolrCloud 模式中，Solr 用来管理集群状态信息的所使用的 Zookeeper 主机的 ip 和端口，如果是 Zookeeper 集群，那么请使用逗号进行分割

<solrcloud> 元素是与 SolrCloud 相关的一些配置参数，除非在 Solr 实例启动时指定了 -DzkRun 或 -DzkHost 参数，否则表 14-4 列举的关于 SolrCloud 的参数将会被忽略。即只有指定了 zkRun 或 zkHost 参数才会开启 SolrCloud 模式，其中 zkRun 参数表示使用 Solr 内嵌的 Zookeeper。

表 14-4 <SolrCloud> 参数表

属 性	描 述
distribUpdateConnTimeout	用于设置分布式更新请求的 Http 连接超时时间
distribUpdateSoTimeout	用于设置分布式更新请求的 Socket 等待超时时间
host	当前 Solr 实例的主机 IP 或域名
hostContext	Solr 实例的服务名称，即 http://ip:port/\${hostContext}
hostPort	当前 Solr 实例的监听端口号，默认取的是 \${jetty.port:8983} 即 Jetty 端口号
leaderVoteWait	当一个 leader 挂掉之后，需要从其他 replica 中重新选举一个 leader 出来，在开始新一轮选举投票开始之前，需要等待的时间，单位：毫秒
leaderConflictResolveWait	Shard Leader 选举过程中，各节点状态信息发生冲突的解决处理的最长时间，默认值为 180 000（毫秒）。当你的 SolrCloud 中有成千上百个小 Collection 时，你可能需要适当加大此参数值
zkClientTimeout	连接 Zookeeper 服务器的超时时间，默认值 30 000，单位：毫秒
zkHost	在 SolrCloud 模式中，Solr 用来管理集群状态信息所使用的 Zookeeper 主机 IP 或域名。当使用的是 Zookeeper 集群时，请使用逗号分隔
genericCoreNodeNames	如果设置为 true，则表示自动生成当前 Core 节点的名称，这样 Core 节点的名称就不取决于 Core 节点的地址，而是一个由 Core 标识的通用的名字。当不同的机器接管这一服务时，这样的名字会更容易被理解。默认生成的 Core 节点名称大致是这样子的：coreNodeName=core_node13

<shardHandlerFactory> 元素用于配置 ShardHandler 实现类的工厂类，由工厂类来生成 ShardHandler 实例，ShardHandler 主要用于处理 SolrCloud 的分布式查询请求。Solr 内置只提供了 HttpShardHandler 这一种实现，是 solr.xml 默认已经显式帮用户配置好了。关于

HttpShardHandler 类中可以配置的属性在 14.2.7 节中已经详细列举过了，这里不再赘述。如果要自定义 ShardHandler，那么你需要实现 ShardHandler 接口，重新定义相应的接口方法，同时自定义相应的 ShardHandlerFactory 实现类，然后将其打包成 jar，导入到当前 Core 的 lib 目录下，通过 solrconfig.xml 中的 <lib/> 元素在 Solr Core 启动加载时自动导入 jar 包，最后通过 <shardHandlerFactory> 元素的 class 属性来指定你自己实现的 shardHandlerFactory 类的完整包路径。

14.5.2 zoo.cfg 详解

`${SOLR_HOME}` 目录下的 zoo.cfg 配置文件其实就是 Zookeeper 的配置文件，只有当你指定 zkRun 系统参数方式启动 SolrCloud 时，才会使用到 `${SOLR_HOME}` 目录下的 zoo.cfg 配置文件。因为一般在实际项目中会单独部署一套 Zookeeper 集群，此时我们会直接配置 Zookeeper 安装目录 conf 配置目录下的 zoo.cfg。表 14-5 详细列举了 zoo.cfg 中可以配置的一些重要参数。

表 14-5 zoo.cfg 参数表

属 性	描 述
clientPort	客户端连接 Zookeeper 服务器的端口号，默认值为 2181
dataDir	用于存放 Zookeeper 快照文件的数据目录
dataLogDir	Zookeeper 的事务日志输出目录，当此参数未指定时，会输出到 dataDir 参数指定的目录下，建议为 dataLogDir 单独挂载硬盘以提升 Zookeeper 性能
tickTime	Zookeeper 中的时间配置的最小单元，Zookeeper 中的其他时间配置都是以 tickTime 的整数倍进行配置
preAllocSize	用于设置预先开辟的用于事务日志写入的磁盘空间，默认值 64M
snapCount	每进行 snapCount 次事务日志输出后，会自动触发一次快照，同时创建一个新的事务日志文件。默认是 100 000
minSessionTimeoutmaxSessionTimeout	用于设置 Session 最大超时时间，默认 Session 超时时间范围为 2* tickTime ~ 20*tickTime
autopurge.purgeInterval	用于设置 Zookeeper 自动清理事务日志和快照文件的频率，单位为小时。默认值为 0，即表示不开启，参数值必须是大于 0 的整数
autopurge.snapRetainCount	用于配置自动清理之后，需要保留几个快照文件，默认值是 3
syncLimit	Leader 发送给 Follower 的心跳包如果在 syncLimit 时间内未能收到 Follower 的响应，那么就判定该 Follower 已下线
initLimit	用于限制 Follower 在启动过程中，与 Leader 同步数据必须在 initLimit 时间内完成
leaderServes	默认情况下，Leader 会接受客户端的请求，并提供正常的读写服务。但是，如果你想让 Leader 专注于集群中节点的协调，不提供其他的读写服务，那么你可以将此参数设置为 no
server.x=[hostname]:port1[:port2]	用于配置 Zookeeper 集群中的各节点，x 表示 myid，port1 用于集群节点之间数据通信，port2 用于 Leader 选举投票通信
cnxTimeout	Leader 选举过程中的 Http 连接超时时间，默认值 5 秒

(续)

属 性	描 述
skipACL	如果开启此参数，那么会跳过对所有客户端请求的 ACL 权限检查
jute.maxbuffer	每个 Znode 最大数据量，默认值为 1M。这个参数必须在 Zookeeper 的 server 和 client 端都设置才会生效 (Java 系统属性: jute.maxbuffer)

14.6 SolrCloud 分布式索引

对于客户端而言，在 SolrCloud 模式下进行文档索引并没有什么太大不同。但是你可以为你的客户端程序进行一些优化以提升性能，以及提供对近实时查询进行支持。

对于 Solr Server 端而言，分布式索引已经发生巨大改变。SolrCloud 会根据索引文档确定它所属的 Shard，并将其转发到相应的 Shard Leader 节点上进行索引。

14.6.1 添加索引文档到 SolrCloud

往 SolrCloud 中添加索引文档，可以通过 Solr 的后台管理界面添加，或通过 Solr 提供的 post.jar 进行添加，这些操作都跟单机模式下没什么太大不同。这里我们主要注重学习如何使用 SolrJ 往 SolrCloud 里添加索引文档进行索引。

之前的 SolrJ 章节中，我们已经学习了如何通过 SolrJ 往 Solr 中添加索引文档，在 SolrCloud 模式下，只需要将 SolrClient 实现改为 CloudSolrClient 即可。其他操作步骤基本雷同。创建 CloudSolrClient 实例的演示代码，如下所示：

```
protected static CloudSolrClient createCloudSolrClient(String zkHost) {  
    // 是否只将索引文档更新请求发送给 Shard Leader  
    boolean onlySendToLeader = true;  
    // 指定索引文档属于哪个 Collection  
    String defaultCollection = "books";  
    // Zookeeper 客户端连接 Zookeeper 集群的超时时间，默认值 10000，单位：毫秒  
    int zkClientTimeout = 30000;  
    // Zookeeper Server 端等待客户端成功连接的最大超时时间，默认值 10000，单位：毫秒  
    int zkConnectTimeout = 30000;  
    CloudSolrClient client = new CloudSolrClient(zkHost, onlySendToLeader);  
    client.setDefaultCollection(defaultCollection);  
    client.setZkClientTimeout(zkClientTimeout);  
    client.setZkConnectTimeout(zkConnectTimeout);  
    // 设置是否并行更新索引文档  
    client.setParallelUpdates(true);  
    // 显式设置索引文档的 UniqueKey 域，默认值就是 id  
    client.setIdField("id");  
    // 设置 Collection 缓存的存活时间，默认值为 1，单位：分钟  
    client.setCollectionCacheTtl(2);  
    return client;  
}
```


关于通过 CloudSolrClient 添加索引文档的完整示例，请参考第 14 章中的示例代码。描述索引文档的创建流程大致如下：

首先创建 CloudSolrClient 实例对象，通过 CloudSolrClient 实例构造索引文档更新请求，然后 CloudSolrClient 实例使用文档路由判断当前索引文档属于哪个 Shard。CloudSolrClient 实例构造子请求并行地发送给每个 Shard Leader。Shard Leader 接收到索引文档后，先在本地对索引文档进行索引，然后记录事务日志以及更新索引文档的版本号，再将索引文档以多线程并行的方式发送给同 Shard 的每个 Replica。Shard Leader 并不关心 Replica 是否挂掉，因为 Replica 会启动自动恢复（Recovering）。最后 Leader 会等待接收每个 Replica 的写入成功确认，并汇总每个 Replica 的响应信息，最终返回给 Client。此时只要当前 Shard 还有一个活跃的 Replica，Solr 仍然可以接收更新请求，这样做的目的是为了增加更新请求处理的吞吐量，但是以牺牲数据一致性为代价。Solr 中的 leaderVoteWait 参数能够减缓一定程度的数据不一致性。但是即便 Replica 会等待 leaderVoteWait 时间，如果 Leader 没能恢复，Leader 上的部分写操作仍会丢失。

14.6.2 SolrCloud 里的近实时查询

从前面的章节中我们了解到，Shard Leader 会把更新请求也转发给同 Shard 的其他 Replica，这样做的目的是为了保证在所有 Shard 上执行查询都能返回一致的查询结果，这很大程度上是由于需要添加对近实时查询的支持。Master-Slave 方式不能支持近实时查询，因为 Master 需要每隔一段时间将整个段文件发送给 Slave，这会导致 Slave 上生成很多小的段文件，从而降低 Slave 的查询性能，这也是为什么 SolrCloud 没有选择 Master-Slave 架构的原因。

在 SolrCloud 中启用近实时查询同样是借助索引的软提交机制，即你需要在 solrconfig.xml 中配置 <autoSoftCommit> 启用索引的自动软提交，配置示例如下所示：

```
<autoSoftCommit>
<maxTime>1000</maxTime>
</autoSoftCommit>
```

当执行一个软提交时，为了保证新添加的索引文档能够对任何查询可见，Solr 内部需要新建一个 IndexSearcher 实例。在新的 IndexSearcher 实例创建完成过程中，为了保证查询性能，Solr 需要对新的 IndexSearcher 实例的缓存进行提前预热，即 autowarm 操作。只有预热完毕，新的数据才能被即时查询。但是，这样做的前提是你的缓存预热和缓存查询的执行速度必须要比软提交的频率要快。比如，你每间隔 2 秒执行一次软提交，等于说每间隔 2 秒 Solr 需要新建一个 IndexSearcher 实例，假如每个新的 IndexSearcher 实例的自动预热过程在这 2 秒内未完成，会导致 Solr 抛出“too many open searcher”异常。在 solrconfig.xml 中一般会限制 indexSearcher 的打开个数，配置示例如下：

```
<maxWarmingSearchers>2</maxWarmingSearchers>
```

也就是你需要确保在下一软提交来临之前完成自动预热。由此可见，自动预热的速

度决定了近实时查询的实时性。

尽管近实时查询功能确实很强大，但是你需要确定是否确实需要使用近实时查询。因为使用近实时查询会导致你的缓存经常失效，你需要不停地新建 `IndexSearcher`，新的 `IndexSearcher` 初始化完成后，旧的 `IndexSearcher` 实例对应的缓存就立即失效了，也就是说你的缓存需要不停地在内存之间“搬来搬去”。更进一步说就是你需要频繁地申请内存和释放内存。

14.7 SolrCloud 分布式查询

一旦你对你的索引数据进行了分片，就会有个问题：你必须对所有的 `Shard` 进行查询才能获取到完整的结果集。针对指定的 `Collection` 跨多个 `Shard`（分片）进行查询，就是我们常说的分布式查询。Solr 会根据用户是否传入 `distrib` 参数或者 `shards` 参数或者 `_route_` 参数来判断当前是否为分布式查询。`distrib=true` 表示开启分布式查询。当你运行在 SolrCloud 模式下，`distrib` 参数默认是 `true`。如果你将 `distrib` 设置为 `false`，那么就只是本地索引查询了。

对于客户端用户而言，使用 SolrJ 执行分布式查询与传统的单机模式下，没有什么太大的不同，不同的只是你可以指定 `shards` 参数，即明确指定在哪些 `Shard` 上执行查询，默认会在 `Collection` 下的所有 `Shard` 上查询。下面是使用 SolrJ 执行分布式查询的简单示例：

```
CloudSolrClient client = TestSolrCloudIndex.createCloudSolrClient(ZK_HOST);
SolrQuery solrQuery = new SolrQuery("*:");
solrQuery.setRows(0);
// 显式指定在哪个 Collection 上执行查询，由于我们在创建 CloudSolrClient 实例时，
// 已经通过 setDefaultCollection 设置了默认查询的 Collection，因此这里你可以不用指定 Collection
solrQuery.set("collection", "books");
// 你也可以调用 client.query("books", solrQuery); 第一个参数即表示你想要查询的 Collection，
// 当你想要临时改变查询的目标 Collection 时，你可能会用到
QueryResponse resp = client.query(solrQuery);
SolrDocumentList hits = resp.getResults();
log.info("Match all docs distributed query found " + hits.getNumFound() + " docs.");
// 显式指定在 shard1 这个分片上执行查询
solrQuery.set("shards", "shard1");
resp = client.query(solrQuery);
hits = resp.getResults();
log.info("Match all docs non-distributed query to shard1 found " + hits.getNumFound() + " docs.");
client.close();
```

然而，并不是所有的 Solr 查询功能都能在 SolrCloud 模式下正常工作。尤其是，SolrCloud 的分布式查询有以下几点限制你必须清楚：

- ❑ IDF 是基于本地索引进行统计，而不是所有 `Shard` 上的索引数据。IDF 值会用于索引文档打分。因此，在分布式查询中，索引文档的评分可能会有所偏差。因为默认情

况下,索引文档是随机跨多个 Shard 分布在多个 Solr 节点上,Shard1 上某个 Term 的 IDF 通常是近似于该 Term 在所有 Shard 上的 IDF 值。如果想要更高的统计精确度,那么你需要配置 *ExactStatsCache*。

❑ Solr 中的 Join 查询不支持跨多个 Shard 请求,仅仅支持单个 Shard 多个 Replica 的情况。但是,当你的数据量巨大时,只划分单个 shard 显然会造成 Solr 查询延迟,性能变差。注意,这里说的 Join 查询指的是使用 `{!join fromIndex=xx toIndex=xxx from=pid to=id}` 这种 Join Query Parser 查询语法的查询,不包括 Block Join。Solr 中的 Block Join 支持跨多个 Shard 查询。通过查阅 Solr 6.2.1 版本中的 *ScoreJoinQParserPlugin* 类的源码我们可以得知(如图 14-13 所示),Join 查询依然不支持跨多个 Shard 查询。即便是单机版的 Join 查询仍有以下几点限制:

- from 参数对应的域建议定义为 `type="string"` 且 `docValues="true"`。假如该域的 `docValues=false`,虽然这样也能执行 join 查询,但是会更消耗内存。
- 数字类型的 DocValues 对于 from 参数暂不支持,支持的 DocValues 类型有 `SORTED`、`SORTED_SET`、`BINARY`。
- to 参数对应的域应该设置为 `indexed="true"`,并且建议 `multiValued=false`,即单值域,即便你设置为多值域也会被当作单值域处理。

```

308 private static String findLocalReplicaForFromIndex(ZkController zkController, St
309     String fromReplica = null;
310
311     String nodeName = zkController.getNodeName();
312     for (Slice slice : zkController.getClusterState().getActiveSlices(fromIndex))
313         if (fromReplica != null)
314             throw new SolrException(SolrException.ErrorCode.BAD_REQUEST,
315                 "SolrCloud join: multiple shards not yet supported " + fromIndex);
316

```

图 14-13 Solr6.2.1 中仍然不支持跨多个 Shard 的 Join Query Parser 查询语法

分布式环境下的 Join 查询确实是一个挑战,因为它需要跨多台服务器与大量的索引数据进行比较。但是随着 Solr 版本的不断迭代,应该会慢慢完善。

- ❑ 每个索引文档必须要有 *UniqueKey*。
- ❑ 如果 Solr 发现有 Document ID 重复, Solr 会选择第一个,丢弃后面的。
- ❑ 如果一个索引提交发生在分布式查询的第一和第二阶段之间,索引数据随时可能不同步。这可能会导致刚开始匹配查询,索引数据随时可能改变不匹配查询但是最后仍然被查询到。这种情况很少发生,也只可能发生在单个查询请求中。
- ❑ Shard 的数量受 Http GET 方法的 URI 的字符长度限制,大部分的 Web Server 支持至少 4000 字符,但是很多 Web Server 会限制 URI 的长度来降低受到 DOS 攻击的安全隐患。

14.8 SolrCloud Collection API

在第 13 章当中,我们已经学习了 Solr 中的 Core API,通过 Core API 你能管理你的

Solr Core。同理，SolrCloud 也提供了 Collection API，通过 Collection API，你也可以管理自己的 Collection，比如创建删除 Collection、为 Collection 创建别名，同时还提供了 Shard 分割等功能。

14.8.1 Collection 常用操作 API

1. 创建 Collection

创建一个 Collection 的 API URL：

```
/admin/collections?action=CREATE&name= &numShards= &replicationFactor=
&collection.configName=
```

其中 action=CREATE 表示执行创建 Collection 动作，name 参数表示想要创建的 Collection 的名称，numShards 参数表示需要将当前 Collection 划分成几个 Shard；replicationFactor 参数表示每个 Shard 需要再分成几个 Replica；collection.configName 参数表示当前 Collection 需要的配置文件（比如 solrconfig.xml、schema.xml 等）目录在 Zookeeper 上的名称，这意味着 Collection 的配置文件目录需要提前上传到 Zookeeper。至于如何上传 Collection 的配置文件目录到 Zookeeper，我们在 14.4.8 节已经介绍过了，这里不再赘述。默认 Solr 只会在“活着”的 Solr 节点上创建 Replica，假如你有 10 个 Solr 节点，但是结果只有 1 个 Solr 节点是活着的，这样会导致所有的 Replica 都会在那活着的 1 个节点上被创建。此时你可以额外添加 maxShardsPerNode 参数来限制每个节点至多能创建多少个 Replica，默认值为 1。当你只想创建一个空的 Collection，暂时不进行任何的 Shard 和 Replica 创建时，你需要添加额外的 createNodeSet=EMPTY 参数，后续可以再通过 ADDREPLICA 操作来创建 Shard 和 Replica。当然也可以添加额外的 autoAddReplicas=true 参数，即开启自动创建 Replica，而不用显式地指定需要创建几个 Shard 几个 Replica。

2. 修改 Collection

当创建完一个 Collection，你可能突然发现某些参数设置错了，想要进行修改，比如 Collection 名称敲错。此时你需要使用修改 Collection 的 API，修改 Collection 的 API URL，如下所示：

```
/admin/collections?action=MODIFYCOLLECTION&collection=<collection-name>&<attribute-
name>=<attribute-value>
```

上面的 action=MODIFYCOLLECTION 参数表示执行修改 Collection 操作，collection 参数用来指定你需要对哪个 Collection 进行修改。<attribute-name>=<attribute-value> 用于覆盖 Collection 创建时的一些参数，支持覆盖的参数有：maxShardsPerNode、replicationFactor、autoAddReplicas、collection.configName、rule、snitch。

3. 重新加载 Collection

在创建 Collection 时，我们必须指定 Collection 所需的配置文件，从创建 Collection

的 API 参数中的 `collection.configName` 可以得到这样一个信息：在 SolrCloud 模式下，Collection 所需的配置文件不再是从本地文件系统里加载，而是从 Zookeeper 上加载，即你需要在创建 Collection 之前将配置文件整个目录上传到 Zookeeper，这样通过 Collection API 创建 Collection 时，你就能够通过 `collection.configName` 参数指定的 Zookeeper 上的配置文件目录名称来引用 Zookeeper 上存储的配置文件。即便使用 Solr 提供的 `solr create_collection` 脚本命令创建 Collection 时，你也可以添加一个 `-d` 参数来指定本地配置文件目录，但是不要被表象所迷惑，其实只不过是 Solr 脚本帮你暗中上传到 Zookeeper 而已。因此，当你想要修改 Collection 的配置文件时，比如在 `schema.xml` 中添加一个域，你需要做的是：首先通过我们在 14.4.8 节介绍的方法从 Zookeeper 上下载获取当前服务器上最新的配置文件到本地，然后进行本地修改，最后将修改后的配置文件上传到 Zookeeper 上。跟之前单机版 Solr 下的 Core 类似，当修改了一个 Collection 的配置文件时，你需要重新加载你的 Collection 才能使其立即生效。此时你需要使用到 SolrCloud 提供的重新加载 Collection API。该 API 接口的 URL 如下所示：

```
/admin/collections?action=RELOAD&name=name&async=requestID
```

上面的 `action=RELOAD` 参数表示执行重新加载 Collection 操作，`name` 参数显然就是你重新加载的 Collection 名称。如果你指定的 Collection 名称不存在，Solr 会抛异常。`async` 参数通过指定一个异步请求 ID 来开启异步操作。

4. 删除 Collection

删除一个 Collection 的 API 接口 URL 如下所示：

```
/admin/collections?action=DELETE&name=collectionName
```

`name` 参数表示想要删除的 Collection 名称，并且 DELETE 接口同样支持 `async` 参数。`async` 参数通过指定一个异步请求 ID 来开启异步操作。

5. 创建 Collection 别名

Collection API 中的 `CREATEALIAS` 接口用于为一个或多个 Collection 指定别名，如果指定的别名已经存在，那么默认会覆盖旧的别名。`CREATEALIAS` 接口的 URL，如下所示：

```
/admin/collections?action=CREATEALIAS&name=aliasName&collections=collections
```

`name` 参数表示别名，`collections` 参数表示 collection 名称，可以为多个 Collection 指定一个别名，多个 Collection 名称采用逗号分割。`CREATEALIAS` 接口同样支持 `async` 参数。

6. 删除 Collection 的别名

删除一个或多个 Collection 对应的别名的 API 接口 URL：

```
/admin/collections?action=DELETEALIAS&name=aliasName
```

`name` 参数表示别名，`DELETEALIAS` 接口同样支持 `async` 参数。

7. Collection 备份与恢复

此接口用于将指定的 Collection 以及该 Collection 关联的配置文件备份到指定的共享文件系统，接口 URL 如下所示：

```
/admin/collections?action=BACKUP&name=myBackupName&collection=myCollectionName
&location=/path/to/my/shared/drive
```

name 参数表示备份名称，可以随意定义。collection 参数表示你需要对哪个 Collection 执行备份操作，location 表示你需要将 Collection 备份文件保存到文件系统的什么路径下。

与 Collection 备份操作对应的就是 Collection 备份恢复，即通过此操作你可以根据 Collection 备份接口操作生成的备份文件，创建一个与指定 Collection 拥有相同索引数据和配置文件的 Collection。如果指定的目标 Collection 不存在，则其会自动创建，注意你不能对一个正在进行备份的 Collection 执行 Collection 备份恢复操作。备份恢复生成一个新 Collection 之后，生成的目标 Collection 与原 Collection 拥有相同的 Shard 和 Replica，以及相同的文档路由信息。如果原 Collection 关联的配置文件目录在 Zookeeper 上存在，那么两者会共用一份，否则目标 Collection 会上传本地备份的配置文件到 Zookeeper。此操作接口 URL 如下所示：

```
/admin/collections?action=RESTORE&name=mybackupName&location=/path/to/my/
sharded/drive&collection=myRestoredCollectionName
```

上面的 name 参数即 Collection 备份接口中 name 参数，表示备份名称，两者需要保持一致，location 参数用于指定 Collection 备份文件的存放路径。collection 参数表示备份恢复后生成的目标 Collection 名称。

8. 迁移 Document 至另一个 Collection

MIGRATE 接口用于根据给定的路由 key 将所有 Document 迁移到另一个 Collection 中。原 Collection 会继续拥有原来的数据，但是会发起重新路由请求，然后在 forward.timeout 参数指定的时间内将索引文档路由到新的 Collection 中。当 MIGRATE 操作执行完成之后，用户可以将 Solr 的读写请求切换到新的 Collection 上。split.key 参数指定的路由 key 可能会在原 Collection 与目标 Collection 之间跨多个 Shard。此 MIGRATE 操作是按照一个 Shard 一个 Shard 的单线程方式进行处理。在 MIGRATE 操作过程中可能会创建临时 Collection，但是在执行完成之后它们会被自动删除。

MIGRATE 操作很耗时，因此强烈建议你指定 async 参数启动异步操作。如果 async 参数未指定，则默认为同步操作。建议你将读超时时间设置大一些。当 MIGRATE 操作失败了，请查看 Solr 日志、集群状态。

MIGRATE 操作只支持使用 compositeId 路由的 Collection，并且在 MIGRATE 操作执行期间，目标 Collection 必须不能接收写请求，否则某些写操作可能会丢失，无法保证可靠性。注意，MIGRATE 操作并不会对 Document 执行任何去重操作。这意味着如果目标

Collection 中已经包含了相同 UniqueKey 的 Document, 那么目标 Collection 中将会包含重复的 Document。

MIGRATE 操作接口 URL 如下所示:

```
/admin/collections?action=MIGRATE&collection=&split.key=&target.collection=&forward.  
timeout=60
```

表 14-6 详细列举了 MIGRATE 操作接口所支持的所有请求参数。

表 14-6 MIGRATE 接口参数表

参数 asdfa	描 述
collection	被迁移的原 Collection 名称
target.collection	原 Collection 上的 Document 迁移到的目标 Collection 名称
split.key	路由 key 的前缀, 用于确定 Document 属于哪个 Shard, 比如 Document ID 为 a!123, 那么此时 split.key 应该指定为 a!
forward.timeout	表示原 Collection 内的 Document 必须在 forward.timeout 参数指定的时间内被重新路由转发到目标 Collection 上。默认值是 60 (单位: 秒)
propertyName=value	用于设置 Core 的 properties
async	用于指定一个 Request ID 来启用异步执行, 默认未开启

9. 查看异步操作的状态

上面的很多 Collection API 接口操作都支持添加 async 参数, 通过指定该参数即可开启接口的异步操作。比如我们执行一个 Shard 分割操作, 示例如下:

```
http://linux.yida01.com:8983/solr/admin/collections?action=SPLITSHARD&collectio  
n=books&shard=shard1&async=60000
```

请注意这里的 async 参数, 我们将其参数值指定为 60 000。但是请不要被参数值所迷惑, 看起来好像是接口操作的操作时间, 其实 async 参数值只不过是一个异步请求的 ID, 后续你可以通过查看异步操作状态接口, 传入这个请求 ID, 然后就能随时得到该异步操作的执行状态。这里的 Request ID 就好比你去餐馆吃饭, 点完菜后, 餐馆并不会立即给你上菜, 而是先给你一个号牌, 然后你根据这个号牌取餐。

14.8.2 Shard 常用操作 API

1. 分割 Shard

分割一个 Shard 会将 Shard 分成两块, 每一块为一个子 Shard。被分割的原始 Shard 会继续包含原有的索引数据。当一个 Shard 正在执行 Shard 分割时, 它仍然能够继续接收客户端的查询和索引请求, 但是当 Shard 分割结束之后, 后续的查询请求将会被路由到新 Shard 上。新 Shard 会拥有跟旧 Shard 一样多的 Replica。在 Shard 分割之后, 不需要显式地执行 commit (无论硬提交还是软提交) 操作, 内部会隐式地执行一个软提交, 这样保证了索引文

档能够在新分割的子 Shard 上可见。

SolrCloud Collection API 里的 Shard 分割操作仅仅适用于使用 numShards 参数创建的 Collection，这意味着你的 Collection 需要依赖 Solr 的基于 Hash 的路由机制。这个分割操作是通过将原始 Shard 的 Hash 范围均分成两个分区，然后根据新的子 Hash 范围对原始 Shard 里包含的 Document 计算 Hash，根据两个分区的 Hash 范围对号入座，从而将 Document 分配到两个子 Shard 上。当然也可以通过 ranges 参数对原始 Shard 的 Hash 范围以十六进制的形式进行任意指定，例如，原始 Shard 的 Hash 范围为 0 ~ 1500，当你添加了参数 ranges=0-1f4,1f5-3e8,3e9-5dc，原始 Shard 就划分成 3 个 Shard，Hash 范围区间依次是 0 ~ 500，501 ~ 1000 和 1001 ~ 1500。此外还可以添加 split.key 参数指定路由 key 来分割 Shard，这样包含该路由 Key 的 Document 会落入该子 Shard 内。当你指定了 split.key 参数，就没有必要再指定 shard 参数，因为根据路由 key 已经足够确定所属 Shard。通过 split.key 参数指定的路由 key 不支持跨多个 Shard。比如，假设 split.key=A! 经过 Hash 计算之后属于 12 ~ 15 这个 Hash 范围，并且属于 Shard1，但是 Shard1 的 Hash 范围是 0 ~ 20，那么根据 split.key=A!，原始 Shard1 会被分割成 3 个子 Shard，Hash 区间依次是 0 ~ 11，12 ~ 15 和 16 ~ 20。但是注意，根据路由 key 划分的子 Shard 对应的 Hash 范围区间匹配到的 Document 可能会与其他路由 key 最终匹配到的 Document 重叠。

如果 Shard 分割之后不可见，你可以尝试重新加载 Collection。另外，Shard 分割操作是一个很耗时的操作，为了避免操作超时，建议以异步方式进行调用。关于异步调用，具体请看本章节的后续介绍。

Shard 分割 API 接口 URL 如下所示：

```
/admin/collections?action=SPLITSHARD&collection=collectionName&shard=shardID
```

上面的 action=SPLITSHARD 表示执行 Shard 分割操作，collection 参数表示你需要哪个 Collection 下的 Shard 进行分割，shard 参数表示你需要对哪个 Shard 进行分割。此外你还可以指定如下可选参数，具体请看表 14-7 的详细描述。

表 14-7 shard 分割 API 接口的可选参数

参 数	描 述
ranges	以十六进制表示的 Hash 范围区间，多个采用逗号分隔，比如 ranges=0-1f4,1f5-3e8,3e9-5dc
split.key	用于分割索引的 key
propertyName=value	用于定义 core 的 properties，可选参数
async	用于指定一个异步请求 ID，当你指定了此参数即表示启用异步执行，建议启用，设置示例 async=60000,60000 即你的异步请求 ID，之后你可以根据这个异步请求 ID 获取 Shard 分割的执行结果，可选参数

2. 创建一个 Shard

创建一个新 Shard 只适用于使用隐式路由的 Collection，对于使用隐式路由的 Collection

而言，创建一个新 Shard 也只有通过此 API 进行创建。如果你的 Collection 使用的是 compositeId 路由，那么你能只能使用 SPLITSHARD 操作 Shard 进行分割来生成新 Shard。下面是创建一个新 Shard 的 API 接口 URL：

```
/admin/collections?action=CREATESHARD&shard=shardName&collection=collectionName
```

表 14-8 详细列举了创建一个新 Shard 的 API 接口支持的所有请求参数。

表 14-8 创建 Shard 的 API 接口参数表

参 数	描 述
collection	用于指定在哪个 Collection 上创建一个新 Shard
shard	用于指定创建的新 Shard 的名称或者说逻辑 ID
createNodeSet	默认 Solr 会在所有“活着”的 Solr 节点上创建 Shard 和 Replica，当你希望在指定的个别 Solr 节点上创建 Shard 和 Replica 时，你需要指定此参数，示例：localhost:8983_solr,localhost:8984_solr,localhost:8985_solr。可选参数
propertyName=value	用于定义 core 的 properties，可选参数
async	见表 14-7

3. 删除一个 Shard

删除一个 Shard 会卸载该 Shard 下的所有 Replica，同时会将其从 Zookeeper 上的 clusterstate.json 中移除，默认还会删除该 Shard 下的每个 Replica 的 instanceDir 和 dataDir 目录。

删除一个 Shard 的 API 接口 URL 如下所示：

```
/admin/collections?action=DELETESHARD&shard=shardID&collection=collectionName
```

表 14-9 详细列举了删除一个 Shard 的 API 接口支持的所有请求参数。

表 14-9 删除 Shard 的 API 接口参数表

参 数	描 述
collection	用于指定想要删除的 Shard 属于哪个 Collection
shard	用于指定想要删除的 Shard 的名称
deleteInstanceDir	是否同时删除每个 Replica 的 Core 目录，默认值 true，可选参数
deleteDataDir	是否同时删除每个 Replica 的数据目录，默认值 true，可选参数
deleteIndex	是否同时删除每个 Replica 下的索引数据，默认值 true，可选参数
async	见表 14-7

4. 强制 Leader 选举

当 Shard Leader 挂掉了，你可以通过此接口操作强制执行一次 Leader 选举从而选出一个新的 Leader。接口 URL 如下所示：

```
/admin/collections?action=FORCELEADER&collection=<collectionName>&shard=<shardName>
```

collection 和 shard 参数表示你需要对哪个 Collection 的哪个 Shard 下的 Replica 进行选择。需要注意的是,只有当正常的 Leader 选举无法正常工作时,你才可能需要执行此接口来强制进行 Shard Leader 选举。但是这个操作会使潜在的导致数据丢失,因为旧的 Leader 上可能会存在新 Leader 上没有的部分更新。

14.8.3 Replica 常用操作 API

1. 创建一个 Replica

通过此接口你可以随时为指定的 Collection 的某个 Shard 下添加一个 Replica。此接口 API 接口 URL 如下所示:

```
/admin/collections?action=ADDREPLICA&collection=collectionName&shard=shardId
```

表 14-10 详细列举了创建一个 Replica 的 API 接口支持的所有请求参数。

表 14-10 创建 Replica 的 API 接口参数表

参 数	描 述
collection	用于指定想要在哪一个 collection 下创建一个 Replica
shard	用于指定想要在哪一个 shard 下创建一个 Replica, 如果此参数未指定, 那么 _route_ 参数必须指定
route	如果指定了 shard 参数, 那么此参数会自动被忽略。此参数用于指定一个 Shard key 来确定此 Replica 属于哪个 Shard
node	用于指定在哪个 Solr 节点上创建该 Replica。节点名称指定示例: linux.yida01.com:8983_solr。可选参数
instanceDir	用于指定创建 Replica 时是否同时创建该 Replica 的 Core 目录, 可选参数
dataDir	用于指定创建 Replica 时是否同时创建该 Replica 的数据目录, 可选参数
propertyName=value	用于指定创建 Replica 时设置该 Replica 的 core.properties, 可选参数
async	指定异步请求 ID, 用于开启异步操作, 可选参数

当你为某个 Collection 下的指定 Shard 创建一个 Replica 时, 可能会提示如下异常:

```
org.apache.solr.common.SolrException:org.apache.solr.common.SolrException:
Cannot create 1 new replicas for collection goods given the current number of live
nodes and a maxShardsPerNode of 6
```

从提示信息中, 我们可以得知, 当前 maxShardsPerNode=6 即每个 Solr 节点上最多有 6 个 Shard, maxShardsPerNode 参数表示每个 Solr 节点上最多可以分布的 Shard 个数, 默认值是 1, 你可以在创建 Collection 时设置此参数。也可以通过修改 Collection 接口更新此参数值。在一个正常的 SolrCloud 集群中, 不容许在同一个活跃的 Solr 节点上出现属于同一个 shard 的多个 Replica。将一个 Replica 添加到一个 Solr 节点上应满足以下条件: numShards*replicationFactor < liveSolrNode*maxShardsPerNode。其中 numShards 表示 Collection 的 Shard 个数, replicationFactor 表示每个 Shard 下的 Replica 个数, liveSolrNode 表示集群中的活跃节点。

2. 删除一个 Replica

此接口用于删除指定的 Replica。如果该 Replica 的 Core 正在运行，那么 Core 会被卸载，然后 Replica 会从集群状态中被移除，默认还会删除该 Replica 的 Core 的 instanceDir 和 dataDir。如果该 Replica 的 Core 或者该 Replica 的所在 Solr 节点挂掉了，那么该 Replica 会从集群状态中删除。如果该 Core 后续又自动恢复了，那么它会自动被卸载。删除一个 Replica 的 API 接口 URL 如下所示：

```
/admin/collections?action=DELETEREPLICA&collection=collectionName&shard=shardID&replica=replicaName
```

表 14-11 详细列举了删除一个 Replica 接口所支持的所有请求参数。

表 14-11 删除 Replica 的 API 接口参数表

参 数	描 述
collection	待删除的 Replica 所属 Collection 的名称
shard	待删除的 Replica 所属 Shard 的名称
replica	待删除的 Replica 的名称
deleteInstanceDir	删除 Replica 同时是否删除 Replica 的 core 目录，可选参数
deleteDataDir	删除 Replica 同时是否删除 Replica 的数据目录，可选参数
deleteIndex	删除 Replica 同时是否删除 Replica 的索引数据，可选参数
onlyIfDown	如果此参数设置为 true，那么如果 Replica 的状态为 ACTIVE，那么将什么都不做。可选参数
async	用于指定一个异步请求 ID，当你指定了此参数即表示启用异步执行，可选参数

3. 将一个节点上的所有 Replica 迁移到另一个节点

此接口 URL 如下所示：

```
/admin/collections?action=RELACENODE&source=source-node&target=target-node
```

上面的 source 参数表示原节点的名称，target 参数表示目标节点的名称。这两个参数是必须指定的，同时你还可以额外的指定 parallel 和 async 参数，parallel 参数如果设置 true 即表示为节点上的每个 Replica 复制都开启一个单独线程并行执行。注意，当 Replica 下的索引数据量很庞大时，此操作可能会导致很大的网络 IO 和磁盘 IO。由于此接口操作并没有对原节点的 Replica 进行加锁，因此在 Replica 迁移过程中请不要执行其他的 Collection 操作。当 Replica 迁移完成之后，原节点上的 Replica 将会自动被删除。

14.8.4 集群管理 API

1. 管理集群的属性

管理集群属性接口可以添加修改删除集群范围内的属性，此 API 接口的 URL 如下所示：

```
/admin/collections?action=CLUSTERPROP&name=propertyName&val=propertyValue
```

表 14-12 详细列举了管理集群属性接口所支持的所有请求参数。

表 14-12 集群管理 API 接口参数表

参 数	描 述
name	属性名称，支持的可选值有 urlScheme、autoAddReplicas、location，如果你设置为其他值，请求会被拒绝并抛出异常
val	属性值，如果属性值指定为空字符串或者 NULL，那么属性会被移除。如果属性已经存在会更新属性值。

2. 查看集群的状态

此接口用于提取集群的状态信息，包括 Collection、Shard、Replica、配置文件、Collection 别名、Solr 节点等几项的状态。此接口的 URL 如下所示：

```
/admin/collections?action=CLUSTERSTATUS
```

此外还可以额外的指定 collection 参数，即返回指定的 collection 相关状态信息，还可以额外执行 shard 和 _route_ 参数，shard 参数用于明确的指定返回哪个 Shard 的相关状态信息，而 _route_ 参数是使用路由 key 类匹配部分 Shard，从而返回符合 _route_ 参数限定的一个或多个 Shard 相关状态信息

14.9 Solr 索引主从复制

所谓 Index Replication（索引复制）就是将 Master 节点上的所有索引数据复制到一个或多个 Slave 节点上。Master 节点用于接收 update 请求，而所有的查询请求有 Slave 节点处理。这种读写分离使得 Solr 能够扩展支持大规模的查询请求。

14.9.1 索引复制简介

Solr 里的基于 Java 实现、通过 HTTP 协议工作的索引复制功能具有以下几点特性：

- ☐ Solr 的索引主从复制不需要借助外部脚本。
- ☐ 只需要在 solrconfig.xml 中进行配置。
- ☐ 除了能够复制索引数据同时支持复制配置文件。
- ☐ 能够使用相同的配置跨平台工作。
- ☐ 不依赖于特定操作系统的文件系统特性，比如硬链接。
- ☐ 与 Solr 紧密集成，Solr 提供了一个后台管理界面来细粒度的全方位的控制 Index Replication（索引复制）。
- ☐ Solr 里的索引复制功能被实现为一个 Request Handler，配置索引复制理论上与配置 Solr 里的其他普通 Request Handler 基本相似。

图 14-14 形象展示了 Solr 中的索引复制的基础架构，Master 的索引会被复制到多个 Slave 上。

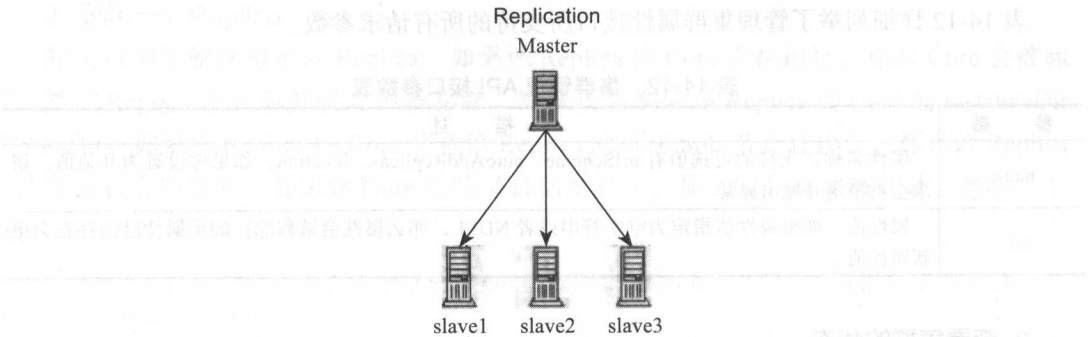


图 14-14 Solr 中的索引主从复制

尽管在 Solr 集群中并没有显式的 Master/Slave（主从）概念，但是 SolrCloud 仍然使用 Replication Handler 来处理 Shard 的恢复。

14.9.2 索引复制的术语

表 14-13 详细列举了与 Solr 索引复制相关的概念术语。

表 14-13 索引复制术语表

术 语	描 述
Index	表示 Lucene 里的索引文件，而这些索引文件由 Solr Core 里的可被搜索和返回的索引数据组成
Distribution	索引的副本从 Master 复制到多个 Slave，这个过程就叫做 distribution。distribution 操作需要利用 Lucene 的索引文件结构
Inserts & Deletes	当对索引执行 insert 或者 delete 操作，索引目录保持不变，索引文档永远是 insert 到一个新建的文件中，索引文档 delete 操作并不是真正的从索引文件中删除。它们仅仅只是在索引文件中将对对应索引文档打个待删除待插入的标记。只有在索引优化阶段才会真正的执行插入或删除
Master & Slave	索引复制中的 Master 是一个单一节点，它初始化接收所有的 Update 请求。索引复制中的 Slave 节点不直接接收 Update 请求，所有的更新（比如 insert、update、delete 等）都是从 Master 节点获取。Master 会将所有索引更新分发给所有 Slave 节点，而所有 Slave 节点只接收来自客户端的查询请求
Update	一个 update 就是针对单个 Solr 实例的单个更新请求，它可能会请求删除一个 Document、添加一个 Document、更新一个 Document 或者删除所有匹配查询的 Document 等。一个 Update 请求会在一个单独的 Solr 实例中被同步处理
Optimization	Optimization 表示会影响索引和段文件合并的一个操作，它应该只在 Master 节点上执行，在索引文件上执行 Optimization 操作会为查询带来性能提升，尤其是当一个索引在一段时间内由于被大量的更新导致生成很多碎片文件。分发一个被优化过的索引需要消耗的时间比分发一个未被优化的索引中新生的段文件要更长
Segments	表示索引的一个自包含子集，它由部分索引文档和那些索引文档中的倒排索引相关的数据结构组成
mergeFactor	用于控制索引中的段文件个数的参数，比如当 mergeFactor=3，那么 Solr 会往一个段文件里填充 Document 直至达到 maxBufferedDocs 参数限制，然后 Solr 会新建一个段文件重新填充其他 Document。当段文件的总个数达到 mergeFactor 参数的限制，此时 Solr 会将所有的段文件合并到一个索引文件中，然后开始将索引文档写入到新的段文件，如此循环

(续)

术 语	描 述
Snapshot	包含索引的数据文件的硬链接的目录。当 Slave 节点主动从 Master 拉取 Snapshots, Snapshots 会从 Master 节点分发到 Slave。Solr 会智能的复制 Slave 节点的 snapshot (快照) 目录中不包含的所有段文件到 Slave 节点

14.9.3 索引复制的配置

下面是 Master 节点上的 solrconfig.xml 中的 ReplicationHandler 配置示例, 设置了一个固定的最大备份数量和一个不变的 maxWriteMBPerSec 参数来方式 Slave 节点的网络接口饱和。

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <str name="replicateAfter">optimize</str>
    <str name="backupAfter">optimize</str>
    <str name="confFiles">schema.xml,stopwords.txt,elevate.xml</str>
    <str name="commitReserveDuration">00:00:10</str>
  </lst>
  <int name="maxNumberOfBackups">2</int>
  <lst name="invariants">
    <str name="maxWriteMBPerSec">16</str>
  </lst>
</requestHandler>
```

在运行一个索引复制之前, 你应该为 ReplicationHandler 设置以下参数 (如表 14-14 所示)。

表 14-14 ReplicationHandler 参数表

参 数	描 述
replicateAfter	可选值有 startup、commit、optimize, 依次表示当 Master 节点启动之后、执行索引 commit 操作之后、执行索引优化操作之后, 可以触发执行索引复制操作。此参数可以配置多个
backupAfter	可选值有 startup、commit、optimize。与 backupAfter 参数类似, 只不过之后触发的是执行索引备份操作。此参数可以配置多个
maxNumberOfBackups	最多保留几个备份文件, 最近生成的前 N 个备份文件将保留, 其余会被删除
confFiles	表示在进行索引复制时, 是否需要顺带复制配置文件, 如果需要, 请指定此参数, 多个配置文件采用逗号分隔
commitReserveDuration	此参数表示每次 commit 之后, 保留增量索引的周期时间, 默认值为 10, 单位: 秒。当你的 commit 操作非常频繁或者网络状况不太好时, 你需要调整此参数。一般将 5M 的数据从 Master 复制到 Slave 需要大约 10 秒, 因此你需要保证 5M 的数据在 commitReserveDuration 参数指定的时间内能够从 Master 复制到 Slave。commitReserveDuration 参数这是粗略估计值
maxWriteMBPerSec	每秒 IO 写速度, 单位: MB, 默认值为 Double.MAX_VALUE
numberToKeep	用于指定保留的备份文件数量, 默认值 Integer.MAX_VALUE, 如果已经指定了 maxNumberOfBackups 参数, 就不能同时指定 numberToKeep 参数, 否则会抛异常

如果你为 replicateAfter 参数配置了 startup，还想要在未来的 commit 或 optimize 操作之后触发索引复制操作，那么你有必要添加 commit 或 optimize。如果你仅仅只为 replicateAfter 参数配置了 startup，那么 commit 或 optimize 操作执行之后不会触发索引复制。

除了可以为 ReplicationHandler 指定上述参数，还可以为其指定一些所有 ReplicationHandler 通用的参数，比如 defaults,invariants。

注意，confFiles 参数还支持对配置文件复制到 Slave 后进行文件改名，示例如下：

```
<str name="confFiles">solrconfig_slave.xml:solrconfig.xml,x.xml,y.xml</str>
```

以上示例表示，Master 节点本地的 solrconfig_slave.xml 文件复制到 Slave 节点上，但是复制到 Slave 节点上之后改名为 solrconfig.xml，而不是 Master 节点上的原名 solrconfig_slave.xml。Slave 节点上的 solrconfig.xml 中的 ReplicationHandler 配置示例如下所示：

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="slave">
    <str name="masterUrl">
      http://linux.yida01.com:8983/solr/replication
    </str>
    <str name="pollInterval">00:00:30</str>
    <!-- 以下是可选参数 -->
    <str name="compression">internal</str>
    <str name="httpConnTimeout">100000</str>
    <str name="httpReadTimeout">1000000</str>
    <str name="httpBasicAuthUser">admin</str>
    <str name="httpBasicAuthPassword">123</str>
  </lst>
</requestHandler>
```


 **注意** 因为 Slave 节点一般会有多个，因此请在其他 Slave 节点也进行 ReplicationHandler 配置。多个 Slave 节点之间的 ReplicationHandler 配置相同。

表 14-15 详细列举了 Slave 节点上的 ReplicationHandler 支持的配置参数。

表 14-15 Slave 节点上的 ReplicationHandler 支持的配置参数表

参 数	描 述
masterUrl	Mster Core 的访问 URL，Slave 通过此 URL 访问 Master 来获取索引数据
pollInterval	表示 Slave 轮询 Master 的间隔时间，即 Slave 每间隔多长时间主动从 Master 拉取最新的索引数据，此参数指定格式为：HH:mm:ss。如果此参数未配置，那么 Slave 不会主动拉取索引保持与 Master 索引同步，但是你可以通过 Solr 的后台管理界面或者 Solr 的 HTTP API 接口来触发 Slave 从 Master 拉取索引
compression	表示在索引传输过程中是否启用数据压缩，可选的值有两个：internal 和 external。如果值是 external，请确保 Master 的 Solr 已经设置了 accept-encoding 头信息。如果值是 internal，索引数据将被自动压缩，这个参数主要在低带宽情况下使用，局域网中请不要使用，局域网中使用会降低索引复制效率。关于 accept-encoding 头信息如何设置请继续往下阅读

(续)

参 数	描 述
httpConnTimeout	用于设置 Slave 连接到 Master 下载索引文件时的最大连接超时时间, 默认值是 5000, 单位: 毫秒, 此参数一般情况下不需要指定, 除非在低带宽情况下或者 Slave 与 Master 节点之间的网络延迟很高时, 你才需要设置此参数
httpReadTimeout	用于设置 Slave 连接到 Master 下载索引文件时 socket 读操作的最大超时时间, 默认值是 10 000, 单位: 毫秒, 此参数一般情况下也不需要指定, 除非在低带宽情况下或者 Slave 与 Master 节点之间的网络延迟很高时, 你才需要设置此参数
httpBasicAuthUser	如果 Master 节点开启了基础的 HTTP 安全认证, 那么 Slave 节点需要配置的用户名和密码
httpBasicAuthPassword	同上

上面提到 compression 参数值如果是 external, 那么需要在 Solr 部署的 Web 容器里配置开启数据压缩。对于 Tomcat 而言, 你只需要在 Tomcat 的 server.xml 添加如下配置即可:

```
<Connector compression="on"
  compressableMimeType="text/html,text/xml,text/plain"
  compressionMinSize="2048"/>
```

而对于 Jetty7 而言, 你需要在 solr 的 web.xml 中添加如下的 filter 配置:

```
<filter>
<filter-name>GzipFilter</filter-name>
<filter-class>org.eclipse.jetty.servlets.GzipFilter</filter-class>
<init-param>
<param-name>mimeTypes</param-name><param-value>text/html,text/plain,text/
xml,application/xhtml+xml,text/css,application/javascript,image/svg+xml</param-
value>
</init-param>
</filter>
<filter-mapping>
<filter-name>GzipFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

如果你使用的是 Jetty6, 那你只需要将上面的 GzipFilter 更换为 org.mortbay.servlet.GzipFilter 即可。

14.9.4 配置索引复制中继器

一个 Master 服务于多个 Slave 在大多数情况下不会有性能影响。但是有可能部署了很多 Slave, 那么每个 Slave 都从 Master 下载索引文件, 这会导致下载操作会占用大量的网络带宽。为了避免类似这种性能问题, 你可以配置一个或多个 Slave 作为中继器, 一个中继器就是一个同时扮演着 Master 和 Slave 角色的节点。

□ 配置一个节点为中继器, 你需要在 solrconfig.xml 的 ReplicationHandler 定义中包含

Master 和 Slave 都会用到的文件。

- ❑ 必须将 `replicateAfter` 参数设置为 `commit`，即便 Master 的 `replicateAfter` 设置为 `optimize`。因为对于中继器或任意的 Slave 而言，Commit 操作只会在索引文件从 Master 节点下载完成之后执行，但是 `optimize` 操作永远不会在 Slave 节点上执行。
- ❑ 可选的，你可以为中继器配置 `compression` 参数，这样中继器中 Master 下载索引文件的时间会相对减少。

ReplicationHandler 中继器配置示例如下所示：

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
<lst name="master">
<str name="replicateAfter">commit</str>
<str name="confFiles">
    schema.xml, stopwords.txt, synonyms.txt
</str>
</lst>
<lst name="slave">
<str name="masterUrl">
    http://master.solr.company.com:8983/solr/core_name/replication
</str>
<str name="pollInterval">00:00:60</str>
</lst>
</requestHandler>
```

14.9.5 索引复制工作机制

对于 Master 而言，当 `startup`、`commit`、`optimize`（索引优化）这其中任意一个操作在 Master 节点上执行完成之后，即表示 Slave 节点可以开始主动 pull 索引文件了。

对于 Slave 而言，Master 节点根本不用关心 Slave 节点，因为 Slave 节点会不断的轮询（依赖于 `pollInterval` 参数）Master 节点，检查 Master 节点上的当前索引文件版本号，如果 Slave 发现 Master 上的版本号比自身的新，那么 Slave 就会启动索引复制操作。索引复制执行步骤依次如下：

- ❑ Slave 向 Master 发送一个 `filelist` 请求，Master 会返回每个文件的名称并作为元数据，元数据包括文件大小、上一次更新时间、别名等。
- ❑ Slave 与自己本地的索引进行比对检查，如果文件在本地索引中存在，然后发送一个 `filecontent` 请求获取缺失的文件。这里使用的是自定义格式来下载文件内容，类似于 HTTP 协议里的 `Chunked` 编码。如果下载过程中 Http 连接中断，下载支持从失败中断点开始继续下载。在任意时刻，在放弃索引复制之前，Slave 会尝试重试 5 次。
- ❑ 从 Master 下载的文件会保存在临时目录，因此在文件下载过程中不管是 Slave 或 Master 崩溃，都不会存在损坏的残留文件存在。相反，当前的索引复制可以简单的中断。
- ❑ 在下载完成之后，所有新文件会被移动到索引目录并且文件的时间戳也与 Master 上

的保持一致。

□ Slave 上的 ReplicationHandler 发送 commit 请求到 Master 节点，同时下载的新索引文件被加载。

想要复制配置文件到 Slave 节点，你需要在 Master 节点上指定 confFiles 参数。只有 Master 节点上的 Solr 实例的 conf 目录下的配置文件会被复制。如果 Master 上的配置文件更新了，该文件只有等到 Master 上一个新的 commit/optimize 操作执行完成之后才能被复制到 Slave。不像索引文件有时间戳就足够判断它们是否一样，配置文件需要依赖文件校验码来判断文件是否更新。当复制配置文件时为了保险起见，在将其移动到索引的 conf 目录下之前，Solr 将配置文件复制到一个临时目录下。之前的旧配置文件然后会被重命名并仍然保存在 conf/ 目录下。Slave 的 ReplicationHandler 并不会自动清理这些旧的配置文件。如果 ReplicationHandler 执行的是配置文件下载操作，那么之后它会发送一个 Core Reload 请求，而不是一个索引提交请求。

对于复制索引文件而言，Slave 如果发现本地的索引文件与从 Master 下载而来的索引文件在文件大小和时间戳方面不相同，意味着 Master 和 Slave 之间的索引文件不兼容。为了解决这个问题，Slave 会将从 Master 下载而来的所有索引文件全部复制到一个新的索引目录，然后请求重新加载 Core 并从新索引目录加载索引。

14.9.6 ReplicationHandler HTTP 接口

你可以使用表 14-16 列举的 HTTP 接口来控制 ReplicationHandler 的操作。

表 14-16 ReplicationHandler HTTP 接口

HTTP 接口	描 述
http://master_host:port/solr/core_name/replication?command=enablereplication	启用 Master 与 Slaves 之间的索引复制
http://master_host:port/solr/core_name/replication?command=disablereplication	禁用 Master 与 Slaves 之间的索引复制
http://host:port/solr/core_name/replication?command=indexversion	查看指定的 Master 或 Slave 上当前索引的最新版本
http://slave_host:port/solr/core_name/replication?command=fetchindex	强制指定的 Slave 从 Master 提取索引副本。你可以传递额外的参数，即你在 <lst name="slave"> 标签下配置的那些参数。这意味着你可以不用在 solrconfig.xml 中将 masterUrl 硬编码，而是在请求 URL 中临时指定
http://slave_host:port/solr/core_name/replication?command=abortfetch	中断指定的 Slave 从 Master 复制索引过程
http://slave_host:port/solr/core_name/replication?command=enablepoll	启用指定的 Slave 轮询 Master 索引更新
http://slave_host:port/solr/core_name/replication?command=disablepoll	禁用指定的 Slave 轮询 Master 索引更新
http://slave_host:port/solr/core_name/replication?command=details	获取配置文件的详情和当前状态

(续)

HTTP 接口	描 述
<code>http://host:port/solr/core_name/replication?command=filelist&generation=<generation-number></code>	获取指定节点上的 Lucene 索引文件, <code>generation</code> 参数值可以通过 <code>indexversion</code> 接口获取
<code>http://master_host:port/solr/core_name/replication?command=backup</code>	在 Master 节点上创建一个备份, 如果没有任何索引数据, 那么什么都不做。此接口可以用于索引周期性备份。此接口支持如下参数: <code>numberToKeep</code> (备份文件保留数量) <code>name</code> : 备份名称, Solr 会在 Core 的数据目录下创建一个名称为 <code>snapshot.<name></code> 的快照文件, 若 <code>name</code> 参数未指定, 那么 <code><name></code> 部分的生成格式为 <code>yyyyMMddHHmmssSSS</code> 。若 <code>location</code> 参数指定了, 那么快照文件就生成在 <code>location</code> 参数指定的路径下而非默认的 Core 数据目录下 <code>location</code> : 备份文件保存路径
<code>http://master_host:port/solr/core_name/replication?command=deletebackup</code>	删除 backup 接口生成的备份文件, 需要传递 <code>name</code> 和 <code>location</code> 两个请求参数。name 即备份名称

14.10 跨数据中心的索引复制 (CDCR)

SolrCloud 架构在某些情况下不太适用, 比如集群节点之间需要昂贵的网络连接开销时。为了防止 SolrCloud 整个数据中心崩溃, Solr 设计了 CDCR 功能特性来缓解这些问题。注意, 这是 Solr6 中的新功能。

14.10.1 什么是 CDCR

CDCR 其实就是两个 SolrCloud 集群之间的索引复制, 这里的数据中心就好比两个 Solr 集群。这里分 Source Data Center 和 Target Data Center, 即源数据中心和目标数据中心, 两者都能接收查询请求, 但是只有源数据中心处理 Update 请求。源数据中心会主动将更新复制到目标数据中心, 这种数据更新复制几乎是实时的, 而且可以配置按照指定间隔周期性发送更新到目标数据中心。CDCR 提前假定两者的起始索引文档是相同的 (当然两者都是空索引也可以)。源数据中心中的每个 Shard Leader 的职责就是复制数据更新到目标数据中心的相应 Collection 中。当目标数据中心的 Shard Leader 接收到更新后, 它会将其复制给它的每个 Replica。这种索引复制模型是设计用来适应一些低带宽或网络连接状况不好的场景以及支持批量更新来优化网络数据通信。当两者是建立在提前建立的索引基础之上, CDCR 只会保证更新的一致性, 不保证整个索引的一致性。因此在 CDCR 搭建之前创建的索引复制需要通过其他方式处理

源数据中心与目标数据中心之间的隐式实现是 push 模型, 因此源数据中心的配置需要知道目标数据中心中的 Zookeeper。CDCR 的索引复制是基于 Collection to Collection 的, 因为 CDCR 是在 `solrconfig.xml` 文件中配置的, 而 `solrconfig.xml` 是为 Collection 量身设计的。

CDCR 还可以配置为在同一个数据中心之间的 Collection 之间进行索引复制。

CDCR 的架构如图 14-15 所示。

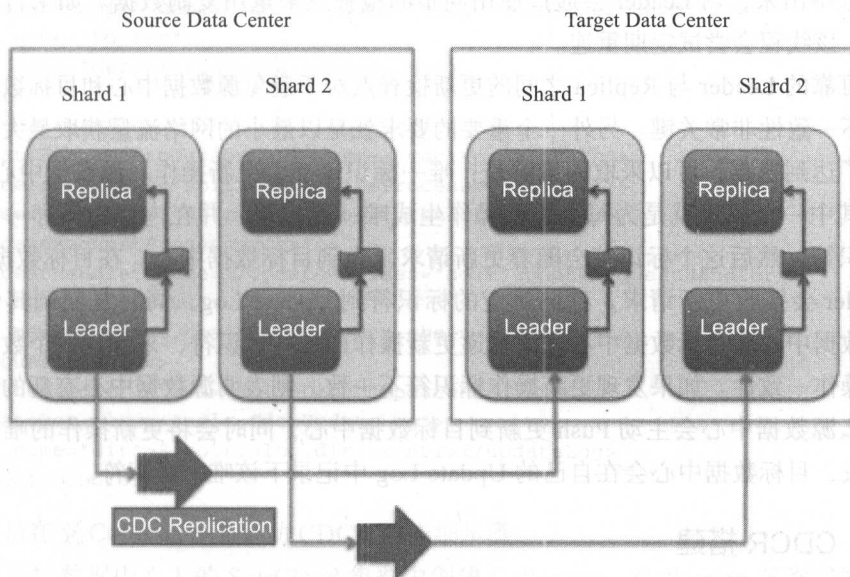


图 14-15 CDCR 架构图

更新和删除首先被写入到源数据中心，然后转发到目标数据中心。其中数据流转顺序如下：

- ❑ 源数据中心的 Shard Leader 接收新的数据更新。
- ❑ 保存到本地索引。
- ❑ 更新的数据在本地索引中保存成功之后会记录 Update Log。
- ❑ 然后将数据更新发送到当前数据中心的同 Shard 的 Replica 上。
- ❑ 当第 4 步成功了，CDCR 从 update log 中读取数据更新，然后 push 到目标数据中心的 Collection。
- ❑ 目标数据中心的 Shard Leader 将数据更新写入到本地索引，然后转发给同 Shard 的所有 Replica。这一步是后台异步执行的。

CDCR 的架构还隐含了一个信息：源数据中心的 Leader 需要知道目标数据中心的 Leader，因为 Leader 可能会改变，同时源数据中心的所有节点需要知道目标数据中心的所有节点，因此防火墙和访问权限规则需要谨慎配置。

14.10.2 CDCR 的 Push 机制

CDCR 的索引复制是借助于 Update Log。CDCR 会起一个后台线程定期去检查整个 Update Log，然后将其转发给目标数据中心。该线程需要根据 Update Log 中上一次成功处

理时间来保存一个检查点，得到目标数据中心的成功处理确认之后，CDCR 会更新 Update Log 里的当前检查点。这个检查点必须跨多个 Replica 同步。当 Leader 挂掉，一个新的 Leader 被选举出来，新 Leader 会通过使用同步的检查点来重用复制数据。如果目标数据中心挂掉了，该线程会尝试定期重连。

一个可靠的 Leader 与 Replica 之间的更新检查点对于避免源数据中心和目标数据中心之间的数据不一致性非常关键。另外一个重要的要求就是以最小的网络流量获取最大化的可扩展性。为了达到这点，可以采取的策略是：唯一标识每一个更新操作。源数据中心的 Shard Leader 的其中一个职责就是为每个更新操作生成唯一标识符，并在内存中保存一份上一次更新的标识符。然后这个标识符会随着更新请求发送到目标数据中心。在目标数据中心端，Shard Leader 会接收更新请求，存储对应的标识符到 Update Log，同时复制到其他 Shard。通过在源数据中心和目标数据中心之间传递更新操作的唯一标识符，来保证两个数据中心之间的更新操作一致性。如果发现更新操作标识符不一致，则表明源数据中心有新的索引数据更新，那么源数据中心会主动 Push 更新到目标数据中心，同时会将更新操作的唯一标识符也发送过去，目标数据中心会在自己的 Update Log 中记录下该唯一标识符。

14.10.3 CDCR 搭建

首先需要搭建好两套 SolrCloud 集群，一套用于源数据中心，一套用于目标数据中心。然后分别为两个数据中心创建一个空的 Collection。

配置源数据中心上的 collection 对应的 solrconfig.xml，配置示例如下所示：

```
<requestHandler name="/cdcr" class="solr.CdcrRequestHandler">
  <lst name="replica">
    <!--注意：这里配置的是目标数据中心使用的 Zookeeper 集群-->
    <str name="zkHost">
      linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181
    </str>
    <str name="source">cdcr_test</str>
    <str name="target">cdcr_test</str>
  </lst>
  <lst name="replicator">
    <str name="threadPoolSize">1</str>
    <str name="schedule">1000</str>
    <str name="batchSize">128</str>
  </lst>
  <lst name="updateLogSynchronizer">
    <str name="schedule">60000</str>
  </lst>
</requestHandler>
<updateHandler class="solr.DirectUpdateHandler2">
  <updateLog class="solr.CdcrUpdateLog">
    <str name="dir">${solr.ulog.dir:}</str>
  </updateLog>
</updateHandler>
```

同理，你需要配置目标数据中心的 Collection 对应的 solrconfig.xml，配置示例如下所示：

```
<requestHandler name="/cdcr" class="solr.CdcrRequestHandler">
<lst name="buffer">
<str name="defaultState">disabled</str>
</lst>
</requestHandler>
<updateRequestProcessorChain name="cdcr-proc-chain">
<processor class="solr.CdcrUpdateProcessorFactory"/>
<processor class="solr.RunUpdateProcessorFactory"/>
</updateRequestProcessorChain>
<requestHandler name="/update" class="solr.UpdateRequestHandler">
<lst name="defaults">
<str name="update.chain">cdcr-proc-chain</str></lst>
</requestHandler>
<updateHandler class="solr.DirectUpdateHandler2">
<updateLog class="solr.CdcrUpdateLog">
<str name="dir">${solr.ulog.dir}</str></updateLog>
</updateHandler>
```

下面是在空 Collection 上启动 CDCR 的详细步骤：

- ❑ 在目标数据中心上的 SolrCloud 集群中创建 Collection，Collection 名称需要与源数据中心上的 solrconfig.xml 中配置的 CdcrRequestHandler 下的 target 属性值保持一致。
- ❑ 启动源数据中心上的 Zookeeper 集群。
- ❑ 启动源数据中心上的 SolrCloud 集群。
- ❑ 同理，上传源数据中心上的 Collection 相关的配置文件到 Zookeeper，你同样需要提前配置好 crRequestHandler 和 CdcrUpdateLog。
- ❑ 在源数据中心上的 SolrCloud 集群中创建 Collection，Collection 名称需要与当前数据中心上的 solrconfig.xml 中配置的 CdcrRequestHandler 下的 source 属性值保持一致。
- ❑ 在源数据中心上执行 START 接口来启动 CDCR（关于 CDCR 的接口稍候再介绍）。
- ❑ 在目标数据中心上执行 DISABLEBUFFER 接口禁用 Buffer 缓冲区。

当你的源数据中心上的 Collection 原始已经有索引数据，此时你启动 CDCR 的步骤会稍微有些不同。下面是具体的操作步骤：

- ❑ 停止源数据中心上的 SolrCloud 中的所有索引创建操作。
- ❑ 停止源数据中心和目标数据中心上的 SolrCloud 集群。
- ❑ 上传源数据中心的 Collection 对应的配置文件到当前数据中心的 Zookeeper 集群。
- ❑ 上传目标数据中心的 Collection 对应的配置文件到当前数据中心的 Zookeeper 集群。
- ❑ 将源数据中心的 Collection 的 Shard Leader 的数据目录复制到目标数据中心的 Collection 的 Shard Leader 的数据目录（目的是为了保持两个数据中心的数据完全一致）。
- ❑ 启动目标数据中心的 SolrCloud 集群。

❑ 启动源数据中心的 SolrCloud 集群。

❑ 在源数据中心上执行 START 接口来启动 CDCR。

❑ 在目标数据中心上执行 DISABLEBUFFER 接口禁用 Buffer 缓冲区。

执行完上述步骤之后, CDCR 就搭建完成了并且能正常工作了。正如你看到的那样, CDCR 的启动过程繁琐且需要纯手工, 这确实是一个问题。但是庆幸的是, 我们只需要操作一次。

然后就可以在源数据中心上往 Collection 里提交一些 Document, 然后你就能在目标数据中心的 Collection 上查询到它们, 因为源数据中心会定期检查本地 Collection 的 Update Log, 而执行索引提交会往 Update Log 里添加日志信息, Update Log 一发生变化, 源数据中心就会自动触发数据复制到目标数据中心, 而这个跨数据中心的索引复制操作几乎是实时的。

14.10.4 CDCR 配置详解

replica 部分的配置即 <lst name="replica"> 标签下的配置, 这部分配置只能在源数据中心端进行配置, CDCR 支持配置多个 replica 参数, 即 <lst name="replica"> 标签可以配置多个。replica 部分配置支持的配置参数, 如下所示:

❑ zkHost: 用于配置目标数据中心的 Zookeeper 集群, 必需指定的参数, 无默认值。

❑ source: 源数据中心中用于索引复制的 Collection 的名称。

❑ target: 目标数据中心中用于接收源数据中心推送过来的索引更新的 Collection 的名称。

CDCR 中的 Replicator 组件的主要职责是转发更新到 Replica (索引副本)。Replicator 还会监控源数据中心上的 Collection 的 Update Log 以及转发新的索引更新到目标数据中心上的 Collection。Replicator 组件使用一个固定的线程池来并行转发索引更新到多个 Replica。如果配置了多个 replica, 那么每个 replica 对应一个线程批量的转发索引更新。replicator 部分支持的配置参数, 如下所示:

❑ threadPoolSize: 用于转发索引更新的线程池大小, 建议一个 replica 配置对应一个线程, 默认值为 2。


❑ schedule: 监控 Update Long 的间隔周期, 单位: 毫秒, 默认值 10。

❑ batchSize: 一个批次内转发到目标数据中心的索引更新数量, 此参数值取决于索引文档的数量。索引文档数量巨大时增大一个批次数量能显著增加你的内存使用率, 默认值为 128。

非 Leader 节点需要时不时地与它们的 Shard Leader 同步 Update Log, 从而清理那些已经过时的事务日志文件。默认这个同步操作是每分钟执行一次, 你可以通过为 updateLog-Synchronizer 部分配置指定 schedule 参数来改变默认的调度行为, 默认值为 60 000, 单位: 毫秒。

CDCR 默认会配置 Buffer 来缓存新的索引更新, 当缓存索引更新时, Update Log 会不断的存储所有索引更新, 对于目标数据中心而言, 根本不需要缓存索引更新, 因为目标数据

中心上根本不存在索引更新操作，因此建议在目标数据中心上禁用 buffer。此时你应该在目标数据中心的 collection 的 solrconfig.xml 中的 buffer 部分配置中指定 defaultState 参数，并将其设置为 disabled。

 **注意** 在配置 CDCR 时，务必在源数据中心和目标数据中心两端的 solrconfig.xml 中都将 Update Log 的实现类配置更改为 solr.CdcUpdateLog。

14.10.5 CDCR 的 HTTP 接口

CDCR HTTP API 用于控制和监控索引复制过程。控制操作都是基于 Collection 级别的，接口的 BASE_URL 为：http://<hostname>:<port>/solr/<collection>，其中监控操作是基于 Core 级别的，其接口的 BASE_URL 为 http://<hostname>:<port>/solr/<core>。

当你想要控制 CDCR，你可以使用如下接口 API：

- ❑ 启动 CDCR (collection_name/cdcr?action=START)。
- ❑ 停止 CDCR (collection_name/cdcr?action=STOP)。
- ❑ 启用用于缓存新的更新的缓冲区 (collection_name/cdcr?action=ENABLEBUFFER)。
- ❑ 禁用用于缓存新的更新的缓冲区 (collection_name/cdcr?action=DISABLEBUFFER)。

当你想要监控 CDCR，请使用如下接口 API：

- ❑ 获取队列中的每个目标和 Update Log 的统计信息 (core_name/cdcr?action=QUEUES)。
- ❑ 获取每个 Replica 上每秒的操作统计信息，这个统计信息基本上能够反映 CDCR 的索引复制性能 (core_name/cdcr?action=OPS)。
- ❑ 获取 CDCR 索引复制过程中的错误信息 (core_name/cdcr?action=ERRORS)。
- ❑ 获取 CDCR 的功能状态信息 (core_name/cdcr?action=STATUS)。

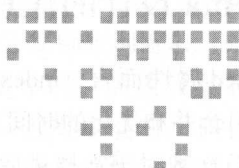
14.10.6 CDCR 存在的限制

CDCR 目前的设计仍然存在一些限制。随着时间的推移，CDCR 也会不断迭代升级，或许不久的将来，这些限制会被一一解决。这些限制如下：

- ❑ CDCR 不适用于索引 Update 频率很高的场景，尤其当源数据中心与目标数据中心之间的网络带宽受到限制时。在这种情况下，初始化时应该执行批量加载，CDCR 使用增量更新来同步源数据中心和目标数据中心。
- ❑ CDCR 目前的 push 机制是源数据中心主动推送索引更新到目标数据中心，目标数据中心完全被动。
- ❑ CDCR 中目标数据中心只能用于数据查询，不能执行索引更新操作，这意味着如果源数据中心整体挂掉，那么我们将无法创建新的索引数据，只能执行索引查询，除非你人工恢复挂掉的源数据中心。

14.11 本章总结

希望大家通过本章的学习都能深刻理解 SolrCloud 并能够熟悉地使用它来处理大规模的索引文档以达到可扩展性、可伸缩性、高可用性、高可靠性。总体来说,我们首先从 SolrCloud 快速入门开始进入 SolrCloud 世界,然后我们在 SolrCloud 工作原理章节理解了 SolrCloud 底层到底是如何工作的。然后我带领大家分别在 Tomcat 和 Jetty Web 容器下搭建了 SolrCloud 集群,随之我们掌握了 SolrCloud 的基本操作,例如如何创建删除 Collection 以及如何管理 Zookeeper 上的配置文件。为了大家更好地配置 SolrCloud,我们又详细了解了 SolrCloud 相关的配置文件。然后我们继续深入学习了 SolrCloud 中如何创建索引以及如何分布式查询。由于大多数情况下操作 Solr 都是借助 SolrJ,我们有必要掌握 SolrCloud 中提供的 Collection API,通过此 API 你能够以编程的方式操作 SolrCloud。虽然 SolrCloud 的架构已经很流行,Solr 索引复制的主从架构仿佛显得被冷落,但是这并不意味着 Solr 索引复制的主从架构模式一无是处了。如果你的项目场景比较适合使用这种主从复制架构,那么你可以选择性地学习该章节的知识点。最后我们还介绍了 Solr 6 中新提出来的 CDCR 架构,它支持 Collection 级别的跨数据中心的索引复制,它缓解了 Solr 主从复制中的一个 Master 服务于多个 Slave 存在的网络性能瓶颈问题,避免 SolrCloud 整个集群挂掉不可用的尴尬。



Solr 性能优化

通过第 15 章，你将可以学习到以下内容：

- Schema 设计的注意事项；
- 索引更新和提交频率的建议；
- Solr 索引合并优化的性能调优；
- 提升 Solr 查询性能的一些优化建议；
- JVM 以及 Jetty/Tomcat 容器的优化；
- 操作系统级别的优化建议。

Solr 性能优化是一个很复杂的任务，也是一个长期与之斗争的过程。本章将带领大家学习应该从哪些方面来对 Solr 进行优化以达到最佳性能。在开始之前，首先需要对影响 Solr 性能的基本因素有个大致的认知。

影响 Solr 性能的一个主要因素就是内存。Solr 需要有足够的内存用于两个方面：一部分用于 Java 堆内存，一部分用于操作系统的硬盘缓存。另外一个潜在的问题就是过高的查询频率，增加内存有时能够使 Solr 处理速度更快。但是如果你想要查询具有可伸缩性，那么最终光靠增加内存是行不通了。此时你需要使用 SolrCloud。

这里我强烈建议你对于 Solr 使用 64 位的 JDK。64 位的 JDK 要求你使用 64 位的操作系统（虽然 32 位的操作系统上也能运行，但性能不是最佳），64 位的操作系统要求 64 位的 CPU。使用 32 位的 JDK 并没有什么不对，主要是使用 32 位的 JDK 会限制你最多只能使用 2GB 作为 Java 堆内存，而 Solr 最需要的就是内存。下面将详细展开 Solr 性能优化的方方面面，希望能给大家一个指导方向。

15.1 Schema 设计的注意事项

对于 indexed 属性而言, indexed=true 的域比 indexed=false 的域在索引时需要占用更多的内存、索引合并和优化的时间更长、索引体积也相应变大,当然如果你不关心 Term 在文档中的出现总次数对于文档最后评分的影响,那么你可以设置 omitNorms=true,它能减少磁盘和内存空间的占用并加快索引创建速度。如果你不需要对该域进行高亮,你还可以设置 omitPositions=true 进一步减小索引体积。如果你只需要利用倒排索引结构根据指定 Term 找到相应的 Document,不需要计算 Term 在 Document 中的出现频率来考虑每个索引文档的权重从而决定其返回的排序,那么还可以设置 omitTermFreqAndPositions=true 即忽略 TF 计算以及 Term 在 Term Vector 中的位置信息,这样能够进一步减小索引体积。需不需要为域设置 indexed=true,这取决于需不需要根据该域进行查询,如果不需要,那么请设置 indexed=false。

对于 stored 属性而言,在响应结果集中通过 fl 参数返回 stored=true 的域的执行开销很大,因为域值需要存储到硬盘,查询时提取域值需要磁盘 IO,因此需要考虑你是否确实需要存储该域值,特别是该域值文本很长很长,比如是一篇几千几万字的文档的内容。但是如果你确实想要在 Solr 查询中能够返回该域值,此时可以考虑使用 ExternalTextField 域,具体如何使用该域请查阅 3.3.5 章节。如果你想要存储的域值长度并不大,但是为了能够缓解提取存储域带来的磁盘 IO,此时可以设置 compressed=true 即启用域值数据压缩。但是请注意,开启域值数据压缩可以降低磁盘 IO,同时会增加 CPU 执行开销。如果并不是一直都需要使用存储域,你可以设置域延迟加载,尤其是当你开启了域值数据压缩。启用域延迟加载,需要在 solrconfig.xml 中进行如下配置:

```
<enableLazyFieldLoading>false</enableLazyFieldLoading>
```

如果你的域值很大很大,比如是一个几十 MB 的 PDF 文件的全部内容,此时如果你将该域的 stored 属性设置为 true 存储在 Solr 索引中,这显然不合适,它不仅会影响你的索引创建性能,还会影响你的查询性能,如果查询时 fl 参数里返回该域那将是雪上加霜。此时可以使用上面提到的 ExternalTextField 域来解决,但是 ExternalTextField 域有一定的限制,比如不能支持 Solr 的查询,只能用于显示和 Function 计算,还可以考虑将域的 stored 设置为 false,而域值存储在外部系统,比如 Memcached、Redis 之类的缓存中,当你需要获取域值时,通过 solr 里的 UniqueKey 去缓存中提取。

对于 tokenized 属性而言,当将一段文本索引到 Solr 的域时,需要考虑后续你是否需要执行类似根据指定关键词查询文档中是否包含该关键字的文档的查询,如果需要,那么此时你就需要设置 tokenized=true 开启对域值进行分词处理。当你的域值本身就是一个简短的单词或者词语,例如表示手机品牌的数据:小米、华为、三星、iphone,对这些数据进行索引就没必要分词了。这并不意味着该域就一定不能分词,如果你的需求是:当用户输入“小”,

能够提示“小米”，输入字母“i”能提示“iphone”，就需要对该域进行 NGram 分词处理。因此，一个域的 tokenized 属性该如何设置，这完全取决于实际查询需求。

对于 multiValued 属性而言，当你的索引数据有多个时，需要设置 multiValued=true 来开启多值域。例如对“鞋子”相关数据进行索引，一般鞋子会有尺码信息，而鞋子尺码一般会有多个，比如：36, 38, 39, 40, 41, 42 等。多值域在功能上会有一些限制，比如 LatLonType 域类型不支持多值域，Solr 中的 Group 查询不支持多值域，Solr 中内置的 ord 函数不支持多值域。因此，当你选择使用多值域时，需要考虑这些限制是否会影响到实际需求的实现。如果这些限制确实制约到你了，此时你可以考虑采用嵌套 Document 来解决此类问题。关于如何创建嵌套 Document，请翻到 12.3.3 节进行学习了解。

对于 Java 里的日期时间类型的数据，建议你使用 Solr 里的 date 域类型，如果你需要进行日期时间范围区间查询，那么建议使用 Solr 里的 date 域类型，而不是使用 string 域类型。

对于 UniqueKey 主键域，建议使用 string 域类型，而不是 int 或者 long。之所以这样建议，是由于以下 3 点理由：

- ❑ Solr 中的 QueryElevationComponent 组件不支持非 string 域类型的 UniqueKey 主键域。
- ❑ SolrCloud 中的 CompositeId 路由要求 UniqueKey 主键域必须为 string 域类型。
- ❑ 比如手机号码如果使用 long 域类型，那么你就不能进行 NGram 处理。

Solr 官方提供的 schema.xml 示例文件中定义了很多 <dynamicField>、<copyField> 以及 <fieldType>，这里建议你将它们都清理掉，<fieldType> 只保留一些基本的 string、boolean、int、long、float、double、date 即可，其他配置后续按需添加即可。但是需要注意，务必保留内置的 _version_ 域，如果你在 solrconfig.xml 中开启了 Update Log，那么 schema.xml 中必须定义 _version_ 域。

15.2 Solr 索引更新与提交的优化建议

Solr 中的索引更新与提交一般不建议显式的调用 commit() 进行硬提交，请在 solrconfig.xml 中配置自动硬 / 软提交。而且在 SolrCloud 模式下，建议每个 Replica 的自动硬 / 软提交的配置保持一致。

此外你应该在 solrconfig.xml 中适当调大 ramBufferSizeMB 和 maxBufferedDocs 参数值，以减缓 Solr 自动触发提交的频率。客户端在提交索引文档时，建议采用批量软提交方式添加索引文档，比如每 1000 个索引文档为一个批次，目的是尽量减少 commit 次数。

在 SolrCloud 模式下，Shard Leader 会将索引更新转发给同 Shard 下的其他 Replica，因此 Shard Leader 与其他各 Replica 之间的网络带宽决定了索引更新的效率。如果该 Shard 下的 Replica 个数太多，那索引更新的性能瓶颈就落在网络 IO 上，因此，此时即便 Shard Leader 与 Replica 之间网络状况很好，由于网络 IO 消耗过多时间同样会影响索引更新性能，此时如果有可能，可以尝试停掉部分 Replica，减少索引更新请求在网络之间进行转发

的次数，从而加快索引更新的速度。停掉部分 Replica 可能不够优雅，甚至可能会导致部分 Replica 节点上数据与 Shard Leader 不一致，此时如果可能的话，你可以尝试再增加几个 Shard，用多个 Shard 去分摊索引更新请求的转发来缓解网络 IO 带来的索引更新性能影响。

对于 Solr 单机模式下，更新或提交索引文档建议你在 Solr 客户端程序中使用 Concurrent UpdateSolrClient 类，对于在 SolrCloud 分布式模式下，建议你使用 CloudSolrClient 类来更新或提交索引文档。

当你进行索引更新或提交时，默认情况下，Solr 会将 Document 的每个域的整个域值进行索引，如果 Document 中的某个域的域值很长很长，因为创建索引过程中 Solr 需要将 Document 缓存在内存中，如果个别域的域值很长，你的内存占用就会很大，内存占用越大，可能会触发更频率的 JVM GC，而 JVM GC 可能会暂停索引创建线程，从而影响索引更新或提交的性能。因此，此时建议你在 schema.xml 中对大文本域使用的域类型配置 LimitTokenCountFilterFactory 来限制实际索引的文本长度，从而减少索引过程中内存占用。

在创建索引时，当需要对文本进行分词处理时，建议你配置停用词来剔除掉无用的噪音 Token，这样既能减小索引体积，同时能够避免噪音 Term 影响最终查询结果。同时，建议你禁用 CompoundFile（复合）文件，虽然它能减少段文件个数，但是它会使得你的索引创建时间增加 7% ~ 33%，一般建议禁用它，请在 solrconfig.xml 中的 <indexConfig> 元素中间添加如下配置：

```
<useCompoundFile>false</useCompoundFile>
<mergePolicy class="org.apache.lucene.index.TieredMergePolicy">
  <float name="noCFSRatio">0.0</float>
</mergePolicy>
```

如果索引创建速度你还是觉得很慢，此时可以考虑借助 MapReduce 分布式处理框架，利用多台机器的资源并行创建 Solr 索引，从而加快索引创建速度。在随书源码的第 13 章中，我有提供一个简单示例（SolrIndexMapReduce 测试类），仅供大家学习参考。

15.3 索引合并性能调优

在索引合并操作中，mergeFactor 参数很大程度上决定了索引合并的频率，虽然索引合并之后能加快 Solr 查询性能，但是索引合并是一个执行开销很大的操作，因此你需要在保证查询性能的前提下，尽量的降低索引合并的频率。索引中的段文件个数增加和每个段文件的体积增大可能会直接触发段文件合并，导致段文件个数增加或段文件体积增大的主要行为就是索引提交。因为每次索引提交都会重新创建一个新的段文件进行索引写入。索引提交除了用户显式的执行 commit 操作之外，ramBufferSizeMB 或者 maxBufferedDocs 参数达到限定的阈值之后也会自动触发索引提交。因此，为了降低索引合并的频率，应该加大 ramBufferSizeMB 和 maxBufferedDocs 参数值，并且尽量降低显式提交的频率，比如采用批量 commit，或者直接在 solrconfig.xml 中配置自动提交并控制自动提交的频率，避免显式提

交。同时，适当加大 `mergeFactor` 参数值来降低索引合并频率。加大 `mergeFactor` 参数值确实可以加快索引创建速度，降低索引合并频率，但是同时它也会降低你的 Solr 查询响应速度。因此，`mergeFactor` 参数是一把双刃剑，你需要在两者之间做好权衡。关于 `mergeFactor` 参数的详细说明，请翻到 3.2 章节进行查阅。不过，我还是要提醒一点，`mergeFactor` 参数的配置在 Solr5.5 版本之后发生了一点细微的变化，Solr5.5 之前 `mergeFactor` 参数配置如下：

```
<mergeFactor>12</mergeFactor>
<maxMergeDocs>345</maxMergeDocs>
<mergePolicy class="..." />
<str name="abc">def</str>
</mergePolicy>
```

而 Solr5.5 版本之后，你需要这样配置 `mergeFactor` 参数，如下所示：

```
<mergePolicyFactory class="...Factory">
<int name="mergeFactor">12</int>
<int name="maxMergeDocs">345</int>
<str name="abc">def</str>
</mergePolicyFactory>
```

下面是一个 Solr5 中索引合并优化后的配置示例，仅供参考。

```
<mergePolicy class="org.apache.lucene.index.TieredMergePolicy">
<int name="maxMergeAtOnce">30</int>
<int name="segmentsPerTier">30</int>
<int name="maxMergedSegmentMB">20000</int>
</mergePolicy>
```

15.4 索引优化的注意事项

你需要定期优化你的索引，因为 Solr 的索引提交是每次生成新的索引段文件，段文件数量越多，会严重影响查询性能。但是索引优化操作的执行开销很大，建议在凌晨 1 ~ 2 点钟再执行此操作。索引优化执行频率与你的索引更新频率相关，如果你的索引数据频繁在更新，那么你需要适当调整索引优化次数。

在 Master/Slave 主从索引复制架构中，如果你在 Master 节点上执行索引优化，将段文件合并为一个，此时 Slave 与 Master 进行数据同步需要复制一个很大的段文件，这会增大索引同步时间，从而会影响 Slave 上的查询响应速度。所以通常一般建议在 Slave 上进行索引优化，而不是 Master 上。

15.5 Solr 缓存

Solr 缓存在 Solr 中扮演着重要的角色，它很大程度上决定了你的 Solr 查询性能。Solr 内置提供了 4 种类型缓存：`filterCache`、`documentCache`、`queryResultCache`、`fieldValueCache`。

Solr 中的缓存都是由 SolrIndexSearcher 实例来管理，一个 SolrIndexSearcher 实例对应一套缓存体系，如果你重新 new 了一个 SolrIndexSearcher 实例，那么之前的 SolrIndexSearcher 实例对应的缓存就全部失效。因此此时你需要对新的 SolrIndexSearcher 实例提前进行缓存自动预热处理，以提升查询性能。

15.5.1 Solr 缓存的常见配置参数

Solr 中的缓存配置是在 solrconfig.xml 的 <query> 元素下进行配置，而这些缓存大都拥有相同的配置属性。

首先 class 属性用于配置 Solr 缓存的实现类，内置提供了 3 种缓存实现：LRUCache、FastLRUCache、LFUCache。当然你也可以选择继承 SolrCacheBase 类进行自定义缓存实现。LRUCache 表示距离上一次命中间隔时间最长的缓存对象优先被剔除。LFUCache 表示一段时间内命中次数最少的缓存对象优先被剔除。FastLRUCache 比 LRUCache 的读取速度更快但是插入速度更慢。size 表示缓存项的最大数目，这里并不是表示内存大小。initialSize 表示初始化能够缓存的项的数目，autowarmCount 表示自动预热过程中从旧缓存中复制的缓存项的最大数目或者百分比（比如 90%），此参数值设置的越大，自动预热的缓存项越多，查询缓存命中率越高，但是同时也会延长自动预热的完成时间，如果自动预热耗时过长可能会造成查询延迟。minSize 参数只适用于 FastLRUCache，是一个可选参数，它表示当缓存项达到 size 大小之后，剔除缓存项直到 minSize 大小，默认值是 $0.9 * size$ 。acceptableSize 参数只适用于 FastLRUCache，是一个可选参数，它表示当剔除缓存项时首先需要尝试删除到 minSize，如果达不到，那么至少尝试删除到 acceptableSize，默认值 $0.95 * size$ 。cleanupThread 参数只适用于 FastLRUCache，是一个可选参数，它表示是否需要单独起一个线程来剔除缓存项，当你的缓存 size 非常大时你可能会需要将此参数设置为 true，默认值 false。timeDecay 参数只适用于 LFUCache，是一个可选参数，它表示当缓存满了，是否需要将旧的缓存项的命中次数按照时间衰变进行减小，以保证最终它们能够被剔除。默认值 true。showItems 参数只适用于 FastLRUCache 和 LFUCache，这是一个用于调试的可选参数，表示缓存项的数目是否需要在统计页面上显示，默认值为 true。

15.5.2 Filter 缓存

FilterCache（即 FilterQuery 缓存）是 Solr 中的顶级缓存，它主要用于缓存 Filter Query 从硬盘上提取出来的 Document 的无序 ID，下次执行同样的 Filter Query 就直接命中缓存。但是只要 SolrIndexSearcher 被重新 open，那么 FilterQuery 缓存就立即失效。以下几个场景会用到 Filter 缓存：

- Filter 缓存会缓存任何 FilterQuery 返回的结果集（实际缓存的是 Document 的 ID 无序 Set 集合），Solr 会显式的将主查询（q 参数）返回的结果集与 Filter 缓存的无序 Document ID Set 集合求交集。

- 尤其是, 当 facet.method=enum 时, Filter 缓存会用于 Facet 查询。
- 如果 solrconfig.xml 中配置了 <useFilterForSortedQuery>true</useFilterForSortedQuery>, 那么对于 Solr 排序操作也会使用 Filter 缓存。
- Filter 缓存通常还会用于其他 Solr 查询, 比如 facet.query、group.query。

如果使用了 facet.method=enum, 建议你将 <filterCache> 的 size 属性值设置的大于 facet 域的唯一值总个数。Filter 缓存的一个配置示例如下所示:


```
<filterCache class="solr.LRUCache" size="16384" initialSize="4096"
autowarmCount="4096"/>
```

至于上面的 size、initialSize、autowarmCount 参数值设置多大, 你需要观察 Solr 后台管理界面里的缓存监控中统计的缓存命中率, 然后不断测试调整达到最优。

15.5.3 Document 缓存

documentCache (即索引文档缓存) 用于存储已经从硬盘上提取出来的 Lucene 中的 Document 对象。Document 缓存的 size 应该大于 <max_results> * <max_concurrent_queries>。<max_results> 表示查询中返回的结果集数量可能的最大值, <max_concurrent_queries> 表示查询的最大并发量, 这样做是为了确保 Solr 查询不需要再从硬盘上提取索引文档。但是你索引文档中的域越多, Document 缓存占用的内存也就越大。

Document 缓存中的 Document 对象包含了多个 Field 引用。如果你设置了 enableLazyFieldLoading=true, 并且开启了 Document 缓存, 那么 IndexReader 提取的 Document 对象就仅仅只包含 fl 参数指定的 Field, 其他 Field 会被标记为“延迟加载”。这样能够减缓 Document 的内存占用, 但是如果延迟加载的域后续被请求到, 那么 IndexReader 会临时从硬盘上加载该域。

 **注意** Document 缓存不能用于自动预热, 因为 Document 缓存使用的是 Lucene 内部的 Document ID, 而当索引数据变化了, 该 ID 也会发生改变, 所以它不能用于新的 IndexSearcher 实例中。

15.5.4 QueryResult 缓存

QueryResultCache (即查询结果集缓存) 用于缓存查询的 Top N 结果集的有序的 Document ID, 按照排序域进行排序。查询结果集缓存的内存占用明显要比 Filter 缓存的小, 因为只有 q 参数、fq 参数、sort 参数同时一致的查询才会命中缓存。

15.5.5 FieldValue 缓存

fieldValueCache (即域值缓存) 与 Lucene 中的 fieldCache 很相似, 但是不同的是

`fieldValueCache` 支持每个 `Document` 对应多个值（多值域的多个域值或者单值域的域值因为分词生成多个 `Term`）。此缓存主要用于 `Facet` 查询。此缓存的 `key` 为域的名称，`value` 为 `docId` 到多个值的映射的数据结构。如果 `solrconfig.xml` 中没有定义 `<fieldValueCache>`，那么 Solr 会自动为你生成一个 `size=10`，`maxSize=10 000`，无 `autowarm` 的 `<fieldValueCache>`。

15.5.6 HTTP 缓存

除了可以在后台服务层启用 Solr 缓存之外，你还可以在前端 HTTP 协议层启用 HTTP 缓存，对于没有更新的资源，可以直接从 HTTP 缓存中直接返回，避免了同样的查询请求频繁请求服务器，这能在一定程度上减轻 Solr Server 的负载压力。在 Solr 中，HTTP 缓存默认并没有开启，如果想要开启 Solr 中的 HTTP 缓存，你需要在 `solrconfig.xml` 中进行如下配置：

```
<httpCaching never304="false">
<cacheControl>max-age=30, public</cacheControl>
</httpCaching>
```

或者：

```
<httpCaching lastModifiedFrom="openTime" etagSeed="Solr">
<cacheControl>max-age=30, public</cacheControl>
</httpCaching>
```

如果两者都有配置，那么以第一种配置为准。`never304` 参数设置为 `false` 即表示开启 Solr 中的 HTTP 缓存，默认 `never304=true` 即禁用 HTTP 缓存。Solr 中的 HTTP 缓存只支持 GET 和 HEAD 请求，不支持 POST 请求。Solr HTTP 缓存兼容 HTTP1.0 和 HTTP1.1 协议头信息。

如果你的 Solr 索引更新频率仅仅是偶尔，比如一小时更新一次，一天一次、一周一次……此时你可以为 `Cache-Control` 请求头添加 `max-age`，`max-age` 的值为索引更新频率的一半，单位：秒。如果你仅仅只需要浏览器缓存来缓存 Solr 的 `Response` 信息，那么你可以为 `Cache-Control` 添加 `private` 设置，示例如下所示：

```
<cacheControl>max-age=30, private</cacheControl>
```

15.5.7 缓存相关的其他配置

你还可以在 `solrconfig.xml` 配置 `firstSearcher` 和 `newSearcher` 事件监听器来自动触发缓存自动预热。`newSearcher` 用于当一个新的 `IndexSearcher` 实例被创建时，除了从旧 `IndexSearcher` 实例自动预热一部分缓存之外，还可以显式的指定一个查询来对缓存进行预热。当某个查询耗时很长时，你可以提前通过 `newSearcher` 监听器进行预热，这样后续你再执行该慢查询时会直接命中缓存。

`firstSearcher` 表示当一个新的 `IndexSearcher` 实例正在被初始化并且当前没有旧的

IndexSearcher 实例用于新的 IndexSearcher 实例进行缓存自动预热, 此时你需要显式的指定一个查询来自动预热缓存。这个 firstSearcher 主要用于配置 Solr 刚启动时执行什么查询并放入缓存。因为 Solr 刚启动时, 缓存肯定是空的, 为了保证刚启动的一段时间内的查询性能高效, 因此你需要配置 firstSearcher 来提取预热。

当使用 queryResult 缓存时, 你还可以额外添加 <queryResultWindowSize> 配置来对其进行优化。当一个查询被执行, 返回的 Document ID 会被收集, 比如查询匹配的 document ID 是 [10, 19] 之间, 如果 queryWindowSize=50, 那么 Document ID [0, 50] 会被收集并缓存, 在此范围内的 Document 将会命中缓存。关于这些配置的详细示例, 请翻到 3.2 章节进行查阅。

15.6 Solr 查询性能优化建议

你的系统可能每天有成千上万的 Document 被添加进来, 每小时有几百几千个查询在执行, 此时查询性能至关重要。索引文档不断增加, 你迟早会遇到堆内存不足的问题, 因此我强烈建议你首先加大你的 JVM 堆内存, 如果你使用的是 Tomcat 容器, 那么请在 catalina.sh 脚本文件中添加如下配置:

```
JAVA_OPTS="$JAVA_OPTS -server -Xms2048m -Xmx2048m"
# OS specific support. $var _must_ be set to either true or false.
```

如果你使用的是默认的 Jetty 容器, 那么请在 solr.in.sh 脚本 (关于 solr.in.sh 脚本的详细介绍请翻到 14.3.2 节进行了解) 中进行如下配置:

```
SOLR_JAVA_MEM="-server -Xms2048m -Xmx2048m"
```

然后, 你需要参照 15.5 节讲解的内容在 solrconfig.xml 中配置 Solr 缓存, 通过观察 Solr Web 后台界面上提供的缓存命中率统计信息对缓存参数不断调整, 保证你的缓存命中率处于一个较高的级别, 缓存命中率越高说明你的 Solr 查询性能越高。具体 Solr 缓存部分如何配置, 这里就不再赘述了。

当你的索引数据量很庞大时, 进行 Facet 查询或者 Sort 排序时会需要大量的内存占用。当然你可以通过增加 Shard 来缓解。然而, Lucene 提供了一种特殊的结构: DocValues, 它使得 Lucene 的 Field Cache 能够运行更快速, 并且能减少内存占用。因此当需要对某个域进行排序或者进行 Facet 查询时, 你可以在 schema.xml 中为域设置 docValues=true, 配置示例如下所示:

```
<field name="title_sort" type="string" indexed="false"
      stored="false" docValues="true" />
```

在 Solr 中默认会为每个 FilterQuery 启用 Filter 缓存, 大部分情况下这能提升查询性能, 并没有什么大问题。但是, 考虑这样一个查询, 比如你想要查询某个价格区间内的书籍,

Solr 查询如下所示：

```
q=solr&fq=category:books&fq=price[50 TO 100]
```

但是对于每个用户而言，设置的价格区间参数可能是不一致的，有的用户提供的价格区间可能是 [20, 50]，也有可能是 [20, 60]、[30, 60]，这种价格区间太多太多，如果对每个价格区间的 Filter Query 都启用 Filter 缓存需要大量的内存来支撑。再比如时间区间查询，如果区间范围精确到秒了，那么此时也不太适合启用 Filter 缓存，比如像下面这个时间区间查询：

```
q=solr&fq=category:books&fq=date:[2016-08-01T13:26:09Z+TO+2016-11-11T10:30:59Z]
```

上面的 Filter Query 的时间区间都精确到秒了，每个用户的时间区间相同的可能性微乎其微，为每个用户这样的时间区间添加缓存显然不合适也不现实。因此，此时你应该为类似这样的 Filter Query 禁用 Filter 缓存，示例如下：

```
q=solr&fq=category:books&fq={!cache=false}date:[2016-08-01T13:26:09Z+TO+2016-11-11T10:30:59Z]
```

至于什么时候应该禁用 FilterQuery 的缓存，你应该考虑的是你的 Filter Query 是否拥有共性，比如 `fq=category:books` 就适合使用 Filter 缓存，因为 `category` 域的域值就固定的几个，用户指定的 `category` 参数值可能不同，但是毕竟 `category` 域值可选值可以预计不会很多，不会达到几百上千个。而且 `category:books` 这类查询用户使用频率高，这样缓存命中率就高，查询性能自然得以提升。比如时间区间查询，由于区间范围不确定性，无法确定对哪个区间的查询进行缓存，即便区间确定了，也很难保证每个用户都会频繁命中这个区间。总之，你要谨记缓存确实可以提升查询性能，但是同时你还要考虑缓存命中率，如果设置一个缓存几天几个月都没有命中几次，那么这是对内存和 CPU 资源的一种浪费行为。

假如你的 Solr 查询中有多个 FilterQuery，此时你需要考虑每个 Filter Query 的执行顺序，因为 Filter Query 的执行顺序可能会影响最终 Solr 查询的性能。考虑下面这样一个查询：

```
q=solr&fq=category:books&fq={!frange l=10 u=100}
log(sum(sqrt(popularity),100))&fq={!frange l=0 u=10}
if(exists(price_a),sum(0,price_a),sum(0,price))
```

这里假定第 2 个 FilterQuery 相比第 1 个 FilterQuery 执行开销更大，且能够过滤的索引文档更少，按照我们在 5.9.3 节讲解的理论，可以得知，应该优先让那些能够过滤掉大部分索引文档的 Filter Query 先执行，而控制 FilterQuery 执行顺序的就是通过设置 `cost` 属性值，`cost` 属性值越小表明其执行开销越小，应该越优先执行。因此，这里我们应该对上面的查询进行如下优化：

```
q=solr&fq=category:books&fq={!frange l=10 u=100 cost=10}
log(sum(sqrt(popularity),100))&fq={!frange l=0 u=10 cache=false cost=50}
```

```
if(exists(price_a),sum(0,price_a),sum(0,price))
```

当需要对数字域进行范围区间查询时，你还可以通过调整 `precisionStep` 属性值来对 Solr 的 Range Query 性能进行优化。假如你需要对 `price`（价格）域进行区间范围查询，`price` 域在 `schema.xml` 中的定义可能类似下面这样：

```
<field name="price" type="float" indexed="true" stored="true"/>
<fieldType name="float" class="solr.TrieFloatField" precisionStep="8" />
```

此时你可以将 `float` 域类型的 `precisionStep` 属性值由 8 改为 4，这样能够大幅度提升针对 `price` 域的 Range Query 性能。至于为什么 `precisionStep` 属性值变小了能够提升 Range Query 性能，请翻到 2.5.2 节进行了解，这里不再重复。但是你需要注意，`precisionStep` 属性值不是越小越好，因为设置越小，你的索引体积也会越大，你应该在两者之间做好权衡。此外修改了域类型的属性，记得重新加载 Core 或者 Collection，然后重建索引。

如果你的查询需要在 3 个域上执行查询，比如 `title:aa content:bbb tag:ccc`，此时你可以考虑使用 `CopyField` 将 `title`、`content`、`tag` 这 3 个域的域值复制到一个新的域上，然后在新域上执行查询，因为在单个域上执行查询比在 N 个域上执行查询性能要高。但是这样做有个弊端，在多个域上分别执行查询你可能单独为每个域设置权重，而采用 `CopyField` 全部复制到一个域上，你就无法实现权重设置。

关于 Solr 查询性能优化还有很多优化点，这里就不再一一展开了，下面就几个我觉得需要提出来的优化点稍作列举，限于篇幅就点到为止了：

- ❑ 前缀查询使用 N-Gram 来实现，而不是使用 `abc*` 的方式进行查询。
- ❑ Phrase Query（短语查询）可以尝试使用 `ShingleFilterFactory` 来提升性能。
- ❑ 你可以使用游标来提升 Solr 分页性能。
- ❑ 查询的时候 `fl` 参数尽量不要返回无关的域，尽量少返回 `stored=true` 的域。
- ❑ 尽量减少 `fq` 的个数，考虑是否可以将多个 `fq` 合并为单个 `fq` 查询。
- ❑ 如果你确定不需要 Solr 中的 NRT 近实时查询，那么请注释掉 `solrconfig.xml` 中的自动软提交配置。
- ❑ 如果你正在使用 Field Facet，那么此时你可以尝试在请求参数中设置 `facet.threads=1000` 来提升 Field Facet 查询的性能，但是此参数只适用于 Field Facet 查询，并且当你的 Solr 查询并发量很大时请不要开启此参数，比如互联网电商项目中请不要使用。
- ❑ 在 SolrCloud 模式下，如果你明确知道你想要查询的 Document 在哪些 Shard 上，那么请显式指定 `shards` 参数或者 `_route_` 参数。
- ❑ 如果有可能，尽量使用最新版本的 JDK 和 Solr。
- ❑ 如果单机已经无法支撑你的业务需求，此时你可以考虑使用 SolrCloud 来解决单机性能瓶颈。

- 进一步考虑业务是否可以拆分，比如原来的音乐、电影查询业务是否可以拆分为两个业务。

15.7 JVM 以及 Web 容器的优化

因为 Solr 是运行在 JVM 实例之上且 Solr 极度消耗内存，因此首先需要分配足够多的内存给 JVM，为 JVM 实例分配堆内存需要使用 `-Xms` 和 `-Xmx` 两个参数。虽然当 Solr 需要的内存超过了 `-Xms` 参数设定值时，`-Xms` 会自动扩容，但是这会导致内存中缓存的对象在堆内存中被来回 Shuffle，而这个 Shuffle 操作可能会影响你 Solr 运行的稳定性。因此一般建议 `-Xms` 参数与 `-Xmx` 参数保持一致即可，尽量减少堆内存自动扩容的次数。`-Xms` 和 `-Xmx` 设置多大，这取决于你的索引数据大小、Solr 需要的缓存大小、创建的 Collection 或 Core 的个数、Facet 查询和 Sort 排序使用频率以及你服务器实际的物理内存大小。如果操作系统发现内存不够用了，操作系统会将内存中的一部分不经常使用的缓存对象溢写到磁盘上的 Swap Space（交换分区），Linux 上的交换分区就好比 Windows 上的虚拟内存，即把磁盘上的一块空间虚拟成一块内存空间。当机器上的物理内存不够用时，操作系统会将内存中不经常使用的缓存对象所占用的内存空间腾出来留给急需内存的应用程序，而腾出来的缓存对象会被溢写到交换分区，其实就是磁盘 IO 写操作。如果应用程序需要使用交换分区里的缓存对象，那么该缓存对象又会从交换分区读入到物理内存。也就是说，当你的物理内存空间不足时，会导致交换分区的磁盘 IO 读写操作，这是为了保证系统应用程序不至于由于物理内存空间不足而无法正常工作，从而采取的挽救之策。但是，应该尽量避免交换分区的磁盘 IO 读写操作的发生。你可以从 Solr 后台的 Dashboard 界面观察到系统 Swap Space（交换分区）的大小以及占用情况，如果你发现 Swap Space（交换分区）被使用了，那么这就表明你的操作系统内存不够用了，该增加物理内存了。

对于 Solr 而言，设置 JVM 最大堆内存大小即 `-Xmx` 参数值的一般经验是：

Max Heap Size = 预期内存中索引大小 + 未来增量添加的索引大小
Min Heap Size = Max Heap Size

JVM 年轻代（即 `-Xmn`）设置建议：

- 年轻代建议设置为大约 Max Heap Size 的三分之一，这里说的年轻代指的是 Eden 区 + 2 个 Survivor Space（幸存区）。
- 建议设置 `-XX:SurvivorRatio=15`，即 Eden 与 Survivor Space 的比例是 15 : 2，由于 Survivor Space 有 2 个，因此三者的比例为 15:1:1。

举个例子，假如你的索引总大小为 7G，你预计未来可能会增加 3G 的索引，此时你可以这样设置：

```
-Xms10G
-Xmx10G
```



```
-Xmn3G
-XX:SurvivorRatio=15
-XX:PermSize=64m
-XX:MaxPermSize=64m
```

关于垃圾回收器的选择，建议使用 CMS 垃圾回收器，配置示例如下：

```
-XX:+UseConcMarkSweepGC           // 启用 CMS 垃圾回收器
-XX:+CMSScavengeBeforeRemark
-XX:+CMSParallelRemarkEnabled
-XX:+UseCMSInitiatingOccupancyOnly
-XX:+ParallelRefProcEnabled
-XX:+ExplicitGCInvokesConcurrent
-XX:ConcGCThreads=24
-XX:CMSMaxAbortablePrecleanTime=6000
-XX:CMSTriggerRatio=80
-XX:CMSInitiatingOccupancyFraction=70
-XX:CMSFullGCsBeforeCompaction=1
-XX:PretenureSizeThreshold=64m
-XX:MaxTenuringThreshold=5
-XX:+UseTLAB
-XX:TLABSize=64K
-Xss256K
-XX:+DisableExplicitGC
```

JDK8 中引入了新的垃圾回收器：G1，G1 是为那些需要大容量内存（一般 6G 以上）和较小 GC 延迟（暂停时间在 0.5 秒以内）的应用程序而设计。相比 CMS 回收器，G1 回收器的优点是：

- G1 是一个压缩回收器，G1 通过充分的压缩完全避免了使用细粒度的空闲列表（free lists）来分配内存，而是使用相对粗粒度的区域（regions）来分配内存。这明显简化了回收器，也很大程度上减少了内存碎片化的问题。
- 相比 CMS，G1 可以预测出垃圾收集的暂停时间，并允许用户指定期望的暂停时间。如果你的应用程序符合以下其中一项或者多项特征，那么从 CMS 或者 ParallelOld 垃圾回收器切换到 G1 可能更合适：

- 活动对象占据了超过 50% 的 Java 堆空间。
 - 对象分配率或者提升率波动明显。
 - 不希望有长时间的垃圾回收暂停时间（暂停超过了 0.5 秒或 1 秒）。
- G1 垃圾回收器的几个比较重要的 JVM 设置参数如下所示：
- -XX:G1HeapRegionSize=n：设置的 G1 区域的大小，值是 2 的幂，范围是 1MB 到 32MB。目标是根据 JVM 的最小堆大小划分出约 2048 个区域。
 - -XX:MaxGCPauseMillis=200：设置期望 GC 暂停的最大时间，默认值为 200，单位：毫秒，但不会严格遵守此设置参数。
 - -XX:G1NewSizePercent=5：设置年轻代占整个堆空间的百分比，默认值为 5 即 5%。

这是一个实验性参数，用于替代旧的 `-XX:DefaultMinNewGenPercent` 设置，此设置在 Java HotSpot VM 中不可用。

- ❑ `-XX:G1MaxNewSizePercent=60`：设置年轻代占整个堆空间的最大百分比，默认值为 60 即 60%。这是一个实验性参数。用于替代旧的 `-XX:DefaultMaxNewGenPercent`。此设置在 Java HotSpot VM 中不可用。
- ❑ `-XX:ParallelGCThreads=n`：设置 STW 期间并行垃圾回收工作线程数，一般设置为 8，如果你的逻辑处理器个数大于 8，可以设置为大约 $(5/8) \times$ 逻辑处理器个数。在大型 SPARC 系统中，可能需要设置为大约 $(5/16) \times$ 逻辑处理器个数。
- ❑ `-XX:ConcGCThreads=n`：设置并行标记线程数，设置为大约并行垃圾回收线程数（即 `ParallelGCThreads` 参数值）的 1/4。
- ❑ `-XX:InitiatingHeapOccupancyPercent=45`：设置当 Java 堆内存占用了多少百分比就开始触发一次标记周期，默认值为 45，即 45%。
- ❑ `-XX:G1MixedGCLiveThresholdPercent=65`：设置老年代占用了整个堆内存多少百分比就开始触发一次混合垃圾回收周期，默认值 65 即 65%。这是一个实验性参数。用于替代旧的 `-XX:G1OldCSetRegionLiveThresholdPercent`。此设置在 Java HotSpot VM 中不可用。
- ❑ `-XX:G1HeapWastePercent=10`：用于设置你愿意浪费的堆内存百分比。默认值是 10 即 10%。此设置在 Java HotSpot VM 中不可用。
- ❑ `-XX:G1MixedGCCountTarget=8`：设置一次全局并发标记之后，最多执行 Mixed GC 的次数。默认值 8。此设置在 Java HotSpot VM 中不可用。
- ❑ `-XX:G1OldCSetRegionThresholdPercent=10`：设置一次 Mixed GC 中能被选入 CSet 的最多 Old Region 占整个堆内存的百分比，默认值 10 即 10%。此设置在 Java HotSpot VM 中不可用。
- ❑ `-XX:G1ReservePercent=10`：设置保留的堆内存空间占整个堆内存空间的百分比，默认值 10 即 10%。此设置在 Java HotSpot VM 中不可用。

如果你想要对 G1 垃圾回收器进行性能调优，那么请记住下面的几条建议：

- ❑ **年轻代大小**：在 G1 垃圾回收器中，避免使用 `-Xmn` 或 `-XX:NewRatio` 等其他相关参数显式设置年轻代大小。固定年轻代的大小会覆盖 `G1MaxNewSizePercent` 参数暂停时间目标。
- ❑ **暂停时间目标**：每当对垃圾回收进行评估或调优时，都会涉及延迟与吞吐量的权衡。G1 GC 是增量式暂停均匀的垃圾回收器，同时应用程序线程的开销也更多。G1 GC 的吞吐量目标是 90% 的应用程序时间和 10% 的垃圾回收时间。如果将其与 Java HotSpot VM 的吞吐量回收器相比较，目标则是 99% 的应用程序时间和 1% 的垃圾回收时间。因此，当评估 G1 GC 的吞吐量时，暂停时间目标不要太严苛。目标太过严苛表示您愿意承受更多的垃圾回收开销，而这会直接影响到吞吐量。当评估 G1 GC

的延迟时,请设置所需的实时(非硬性的)目标,G1 GC 会尽量满足,但副作用是,吞吐量可能会受到影响。

❑ **掌握混合垃圾回收:**当调优混合垃圾回收时,请尝试以下参数(有关这些参数的信息,请参见上面的参数解释说明):

```
-XX:InitiatingHeapOccupancyPercent
-XX:G1MixedGCLiveThresholdPercent
-XX:G1MixedGCCountTarget
-XX:G1OldCSetRegionThresholdPercent
```

如果在生产环境或者测试环境下,你可以开启 GC 的调试参数。打印 GC 过程中的详细信息,供调试使用,可供设置的调试参数如下所示:

- ❑ **-verbose: gc。**
- ❑ **-XX: +PrintGC。**
- ❑ **-XX: +PrintGCDetails。**
- ❑ **-XX: +PrintGCTimeStamps。**
- ❑ **-XX: +PrintGCApplicationConcurrentTime:** 打印每次垃圾回收前,程序未中断的执行时间。可与上面混合使用。
- ❑ **-XX: +PrintGCApplicationStoppedTime:** 打印垃圾回收期间程序暂停的时间。可与上面混合使用。
- ❑ **-XX: PrintHeapAtGC:** 打印 GC 前后的详细堆栈信息。
- ❑ **-Xloggc: filename:** 与上面几个配合使用,把相关日志信息记录到文件以便分析。

Solr 一般部署在 Web 容器中,而常见的 Web 容器一般都支持为同一时间连接的用户并发的创建不同的 Session。

如果你使用的是 Tomcat 容器,那么你可以对 conf 目录下的 server.xml 进行配置调整,<Executor> 元素可以优化的参数项如下:

```
<Executor maxThreads="600" minSpareThreads="100" maxSpareThreads="500" acceptCount="700" acceptorThreadCount="2" maxConnections="6000"
keepAliveTimeout="60000" maxKeepAliveRequests="100"
processorCache="300" socketBuffer="10240" />
```

- ❑ **maxThreads:** request 请求处理最大线程数,maxThreads 过大会增加 CPU 和内存消耗,建议不要超过 6000。
- ❑ **minSpareThreads:** 初始化时创建的 request 请求处理线程数。
- ❑ **maxSpareThreads:** 一旦创建的 request 请求处理线程超过这个值,Tomcat 就会关闭不再需要的 socket 线程。这个配置 Tomcat7.x 里已经不存在。
- ❑ **acceptCount:** 当处理请求的可用线程数被用光时,可以放到处理队列中的请求数,超过这个数的请求将不予处理。
- ❑ **acceptorThreadCount:** 开辟多少个线程来接收 http 请求连接,默认值为 1,可以酌情

加大,但不要超过你的 cpu 核数,具体数字自己测试。

❑ **maxConnections** : 最大接收的 http 连接容量,超出 **maxConnections** 后,可以继续接收 http 连接,但此时放入的连接会被阻塞 http 请求不被处理,直到把 **accpetCount** 队列塞满就不再接收新 http 连接。

❑ **keepAliveTimeout** : 表示在下次请求过来之前, tomcat 保持该连接多久。这就是说假如客户端不断有请求过来,且未超过过期时间,则该连接将一直保持。

❑ **maxKeepAliveRequests** : 表示该连接最大支持的请求数。超过该请求数的连接也将被关闭(此时就会返回一个 **Connection: close** 头给客户端),开启 **keepAlive** 策略可以减少 tcp 握手耗时,从而提高 Http 连接利用率。设置为 1 表示关闭 **KeepAlive** 策略不重用 http 连接,设置为 -1 表示该连接为长连接,永不关闭。**maxKeepAliveRequests** 设置过大会占用系统资源,请酌情设置。

❑ **processorCache** : 用于缓存 http 请求处理对象,提高请求处理性能,默认值是 200,建议将这个值与 **maxThreads** 值设置的一样。

❑ **socketBuffer** : 设置 socket 输出流缓冲区大小,默认值是 9000。

<Connector> 元素可以优化的参数项如下所示:

```
<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
connectionTimeout=20000 compression="off"
enableLookups="false" disableUploadTimeout="true" server="Neo App Srv 1.0" />
```

❑ **protocol** : 设置为 **Http11NioProtocol**, 开启 NIO 工作模式。

❑ **connectionTimeout** : 表示 http 链接最大超时时间,超过限定的时间仍未连接到 Tomcat server 则视为网络链接失败。默认值 60 000,单位:毫秒,设置为 0 表示永不超时,这样设置有隐患的。通常可设置为 30 000 毫秒。

❑ **compression** : 如果你的系统中使用了 apache server 或者 nginx,那么此时请关闭 tomcat 输出流压缩,减轻 tomcat 的负担,将输出流内容压缩工作交给它们来做。

❑ **enableLookups** : 是否启用 tomcat 域名查询,建议禁用,加快 tomcat 的响应速度。

❑ **disableUploadTimeout** : 是否禁用上传超时时间。

❑ **server** : 用于设置 Server 的说明信息,建议自定义并隐藏 tomcat 版本信息,泄漏 tomcat 版本信息,可能会被人攻击。

如果你是直接将 solr.war 包放置在 Tomcat 的 webapps 目录下,那么建议你将 **autoDeploy** 属性设置为 **false**。因为自动部署会导致 CPU 占用很高,可能还造成内存泄漏。配置示例如下所示:

```
<Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="false">
```

如果你的 Solr 是部署在 Jetty 容器中,那么此时需要修改 solr-6.2.1\server\etc 目录下的 jetty.xml 配置文件。首先是配置 **ThreadPool** 线程池,下面是一个配置示例:

```

<Set name="ThreadPool">
<New class="org.eclipse.jetty.util.thread.QueuedThreadPool">
<Set name="minThreads">50</Set>
<Set name="maxThreads">500</Set>
<Set name="detailedDump">false</Set>
</New>
</Set>

```

优化时你可以调整最小线程数 `minThreads` 和最大线程数 `maxThreads`。最大线程数不要超过 500。`detailedDump` 表示是否记录详细的 thread dump，默认不记录。

然后对 Jetty 的 Connector 连接器配置进行优化，Jetty 中的 Connector 默认实现是，Connector 的配置示例如下所示：

```

<Call name="addConnector">
  <Arg>
    <New class="org.eclipse.jetty.server.nio.SelectChannelConnector">
      <Set name="host"><Property name="jetty.host" /></Set>
      <Set name="port"><Property name="jetty.port" default="8983" /></Set>
      <Set name="maxIdleTime">3000</Set>
      <Set name="acceptors">8</Set>
      <Set name="statsOn">false</Set>
      <Set name="confidentialPort">8443</Set>
      <Set name="lowResourcesConnections">10000</Set>
      <Set name="lowResourcesMaxIdleTime">1000</Set>
      <Set name="acceptQueueSize">1000</Set>
    </New>
  </Arg>
</Call>

```

❑ `host`：Jetty 所在主机的 IP 或域名。

❑ `port`：设置 Jetty 的启动端口号。

❑ `maxIdleTime`：表示连接最大空闲时间，默认值 300 000，单位：毫秒，这个值太大，设置为 3000 左右即可。

❑ `acceptors`：设置接收客户端 Http 连接的线程数量，Acceptor 的功能是接收客户端连接然后分配个给 ThreadPool 处理，表示同时在监听 read 事件的线程数，默认值为 2，对于 NIO 来说，建议值 $2 * (\text{CPU 核数} - 1)$ 。

❑ `acceptQueueSize`：在操作系统发起拒绝连接之前，请求连接都保存在队列中，这里表示队列中允许排队的元素数量。

❑ `statsOn`：是否开启统计功能，正式运行环境下建议关闭统计功能。

❑ `confidentialPort`：受信任的端口号。

❑ `lowResourcesConnections`：连接数量达到该数值时，Jetty 会认为服务器资源已经快耗尽了，进入低资源状态。只有 NIO 才有这个设置，表示连接空闲时的最大连接数，大于这个数值将被 shutdown，每个 acceptor 的连接数 = $(\text{lowResourcesConnections} + a$

ceptor - 1) / acceptor。

❑ **lowResourcesMaxIdleTime**：表示可用线程稀少时或者当资源饱和时，连接最大等待时间，单位：毫秒，一般设置为 $\leq \text{maxIdleTime}$ 。

为了最优公平的处理服务器上每个用户的请求，你有必要对每个用户连接占用的系统资源进行限制，从而保证服务器拥有最大化的吞吐量，确保整个服务器运行在限制的资源范围内。此时，你需要为 Jetty Server 添加 LowResourcesMonitor（低资源监视器）来监视系统的低资源情况以及限制服务器上的空闲连接的数量。LowResourcesMonitor 的配置示例如下所示：

```
<Call name="addBean">
  <Arg>
    <New class="org.eclipse.jetty.server.LowResourceMonitor">
      <Arg name="server"><Ref refid='Server'/></Arg>
      <Set name="period"><Property name="jetty.lowresources.period" deprecated="lowresources.period" default="1000"/></Set>
      <Set name="lowResourcesIdleTimeout"><Property name="jetty.lowresources.idleTimeout" deprecated="lowresources.lowResourcesIdleTimeout" default="1000"/></Set>
      <Set name="monitorThreads"><Property name="jetty.lowresources.monitorThreads" deprecated="lowresources.monitorThreads" default="true"/></Set>
      <Set name="maxConnections"><Property name="jetty.lowresources.maxConnections" deprecated="lowresources.maxConnections" default="0"/></Set>
      <Set name="maxMemory"><Property name="jetty.lowresources.maxMemory" deprecated="lowresources.maxMemory" default="0"/></Set>
      <Set name="maxLowResourcesTime"><Property name="jetty.lowresources.maxLowResourcesTime" deprecated="lowresources.maxLowResourcesTime" default="5000"/></Set>
    </New>
  </Arg>
</Call>
```

如果 **monitorThreads** 配置为 **true** 并且 **Executor** 实例为 **ThreadPool**，然后低资源监视器会调用 **isLowOnThreads** 方法来检查当前系统是否处于低资源状态。

如果 **maxConnections** 参数配置为大于零且连接总数超过 **maxConnections**，那么此时低资源监视器判定操作系统进入低资源状态。

如果 **maxMemory** 参数配置为大于零且 JVM 的堆内存减去 JVM 的空闲内存超过 **maxMemory**，那么此时低资源监视器判定操作系统进入低资源状态。

一旦操作系统进入低资源状态，低资源监视器会迭代所有已存在的 **Http** 连接并且设置它们的最大空闲时间为配置的 **lowResourcesIdleTimeout** 参数值。这样有助于空闲线程快速被回收。

如果操作系统的低资源状态持续的时间超过了配置的 **maxLowResourcesTime** 参数值，那么低资源监视器会反复应用 **lowResourcesIdleTimeout** 参数来限制新的 **Http** 连接。

最后，建议将 JVM 设置为 **Server** 工作模式来提升性能，你只需要在 Solr 启动参数中添加

加 `-server` 即可，与设置 `-Xms`、`-Xmx` 参数类似，具体如何设置请翻到 15.6 章节进行查阅。

15.8 操作系统级别的优化建议

操作系统和硬件从根本上决定了系统的性能。Solr 使用标准的文件系统来存储索引数据，而且 Solr 的所有操作都依赖底层的操作系统 API。为了使得服务器能够支持更多的 TCP/IP 连接以及更高的系统吞吐量，你需要对操作系统进行适当的调优。注意：这里只讨论 Linux 系统的优化。

Linux 系统的 `/etc/security/limits.conf` 配置文件中可以被限制的系统资源包括：`core`（限制内核文件的大小）、`data`（最大数据大小）、`fsize`（最大文件大小）、`memlock`（最大锁定内存地址空间）、`nofile`（打开文件的最大数目）、`rss`（最大持久设置大小）、`stack`（最大栈大小）、`cpu`（以分钟为单位的最多 CPU 时间）、`noproc`（进程的最大数目）、`as`（地址空间限制）、`maxlogins`（此用户允许登录的最大数目）。`limits.conf` 的配置格式如下：

```
<domain><type><item><value>
```

`<domain>` 可以是一个系统用户名称，也可以是一个用户组名称，采用 `@` 用户组名称语法格式，可以是星号表示系统所有。`<type>` 可选值有：`soft` 和 `hard`，`soft` 设置起一个警示作用，`hard` 类型才是硬性限制。`<item>` 就是被限制的资源，比如 `fsize`、`nofile`、`cpu`、`core` 等。

由于 Solr 中每次索引提交都会重新创建一个段文件用于索引写入，如果你的索引提交频率很频繁，那么你可能会遇到“Too many open files”异常。很多操作系统默认能够打开的文件个数为 1024 个，因此你需要调大这个限制，此时你可以在 `/etc/security/limits.conf` 配置文件中设置 `nofile` 限制，配置示例如下所示：

```
root          hard    nofile      100000
@solr         hard    nofile      100000
```

建议调大 Linux 系统用户的最大可创建进程数，编辑 `/etc/security/limits.d/90-nproc.conf` 配置文件进行调整，配置示例如下：

```
*              soft    nproc       1024
root           soft    nproc       unlimited
hadoop         soft    nproc       unlimited
```

你还应该将 TCP 的缓冲区调大到至少 16M，配置示例如下所示：

```
sysctl -w net.core.rmem_max=16777216
sysctl -w net.core.wmem_max=16777216
sysctl -w net.ipv4.tcp_rmem="4096 87380 16777216"
sysctl -w net.ipv4.tcp_wmem="4096 16384 16777216"
```

默认连接监听队列大小为 128，如果你正在运行的服务器负载很高，并且开始出现 TCP

级别的连接被拒绝问题,那么此时你需要增大此参数值。如果将此参数设置的过大,大量的 TCP 连接会耗费大量的系统资源,如果设置的过小,那么会出现 TCP 连接被拒绝,配置示例如下:

```
sysctl -w net.core.somaxconn=4096
```

`net.core.somaxconn` 用于控制用于上层 (Java) 处理的传入数据包队列大小。默认值是 2048,你可以加大此参数值,以及调整相关参数:

```
sysctl -w net.core.netdev_max_backlog=16384
```

```
sysctl -w net.ipv4.tcp_max_syn_backlog=8192
```

```
sysctl -w net.ipv4.tcp_syncookies=1
```

如果你创建了很多连接,比如 `load generator` (负载生成器),此时操作系统在端口号上运行会很慢,因此你最好是增加端口号的范围,并且允许重用 `TIME_WAIT` 状态的 `socket`。

```
sysctl -w net.ipv4.ip_local_port_range="1024 65535"
```

```
sysctl -w net.ipv4.tcp_tw_recycle=1
```

TCP 拥塞控制算法对 TCP 传输速率的影响可很大。丢包使得 TCP 传输速度大幅下降的主要原因就是丢包重传机制,控制这一机制的就是 TCP 拥塞控制算法。执行如下命令查看当前系统内核中可用的 TCP 拥塞控制算法有哪些:

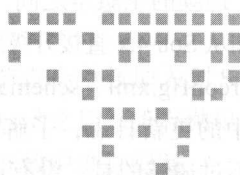
```
sysctl net.ipv4.tcp_available_congestion_control
```

如果 `cubic` 或 `htcp` 没列举出来,那么你可以尝试设置为 `cubic`,如下所示:

```
sysctl -w net.ipv4.tcp_congestion_control=cubic
```

15.9 本章总结

在本章中,我们介绍了多种不同的方式来优化 Solr 实例的索引与查询的性能,通过这些措施能够让你的 Solr 运行得更快更稳定,同时,建议你通过一些性能监控工具来实时监控你的 Solr 服务,便于及时发现问题并予以解决,保证你的 Solr 能够持续良好运行。在第 10.9.3 章节我们介绍了如何使用 Solr 自身提供的 JMX 特性来监控 Solr 性能,这里我再推荐一款开源的监控工具: `Ganglia`(<http://ganglia.info/>),至于如何使用它来监控 Solr,就留给大家自己学习研究了。



Solr 扩展篇

通过第 16 章，你将可以学习到以下内容：

- ☐ Solr 如何版本升级；
- ☐ Solr 中如何自定义伪域；
- ☐ Solr 多语种索引支持；
- ☐ Solr 中如何自定义 Redis 缓存；
- ☐ Solr 如何开启 HTTPS；
- ☐ Solr 安全认证；
- ☐ SolrCloud 模式下的索引增量更新；
- ☐ Solr 与 MapReduce 集成；
- ☐ Solr6 中的 SQL 接口；
- ☐ Solr6 中的流式处理；
- ☐ Solr 与 HBase、Flume、Storm、Kafka、Spark 的集成。

在本章中，我们将继续学习 Solr 中的一些扩展知识，也是很多同学急切关注的问题，此外还会额外介绍一些 Solr 6 中新添加的功能特性，比如 Solr 6 中的 SQL 接口以及流式处理等。最后我会带领大家学习 Solr 与大数据领域框架之间如何交互。

16.1 Solr 如何版本升级

如果你想要从 Solr 3.x 升级到 Solr 4.x 或者任意两个主版本之间升级，比如 4.x → 5.x、5.x → 6.x。那么在升级过程中需要记住以下几点：

- Solr 两个连续的主版本之间，索引格式一般支持向后兼容，但是如果你要跨主版本升级，比如从 Solr 4.x 直接升级到 Solr 6.x，那么此时你先升级到 5.x，再从 5.x 升级到 6.x。
- 你的 solrconfig.xml、schema.xml 以及 solr.xml 需要手动编辑，这意味着你需要阅读新版本中的更新日志，了解哪些配置已经被标记为废弃或者已经被移除已经替代的方案。不过遗憾的是，没有可选的工具能够帮助你自动完成。
- 确认你已经修改了 solrconfig.xml 中的 luceneMatchVersion 属性配置。
- 自 Solr4.8 版本开始，要求必须最低 JDK7+，自 Solr5.5 版本开始，要求必须 JDK8+。
- 如果你成功升级了，你最好尝试下对你的 Solr 索引执行下索引优化操作，确保索引能够正确重写。
- 如果你使用的是 SolrCloud，你需要更新 solr.in.sh 的配置，然后重新执行 solr 的 bin 目录下的 install_solr_service.sh 安装脚本。
- 如果你发现索引格式不兼容，那么很遗憾，你需要全部重建索引。

进行 Solr 版本升级，还会波及你的 SolrJ 客户端程序，因为一般 Solr API 会发生更新，比如一些类可能会被废弃，比如 SolrServer → SolrClient。你需要查阅 Solr 的更新日志根据提示将其替换为最新版本的使用方式。

下面着重介绍如何升级 SolrCloud 集群，在开始升级之前，需要准备以下工作：

- 如果你的 SolrCloud 中只有单个 Replica，那么你最好备份下每个 Collection。
- 升级时，建议集群中每个节点挨个启动，即等一个启动成功之后再启动第二个。
- 确保你当前的集群状态是正常的，并且所有的 Replica 都处于 ACTIVE 状态。
- 如果你的 solrconfig.xml 中包含了自定义组件，那你需要首先根据新的 Solr API 对你的自定义组件进行 API 版本升级。
- 确认以下 Solr 启动脚本中需要使用到的参数：
 - ZK_HOST: Zookeeper 集群的连接字符串。
 - SOLR_HOST: 当前主机的 ip 或域名。
 - SOLR_PORT: Solr 实例监听的端口号，默认 8983，确认端口号没有被占用。
 - SOLR_HOME: Solr Core 的宿主目录，该目录下必须包含 solr.xml。

OK，下面可以开始进行 SolrCloud 升级了：

第一步：先停掉你想要升级的 Solr 节点。

第二步：修改 solr.in.sh 脚本配置。

如果你之前已经安装过 Solr，那么 Solr 会将 solr.in.sh 脚本复制到 /etc/default/ 目录下，你需要复制一份，然后对其中一些安装过程中需要使用到的参数进行修改，具体请参照 14.3 章节。

第三步：安装 Solr 服务。

请按照 14.3 章节介绍的步骤重新安装 Solr 服务，在安装之前，请确保当前用户有执行 install_solr_service.sh 脚本的权限，以及当前用户拥有对你在 solr.in.sh 脚本中配置的 SOLR_

HOME 目录的写权限。

第四步：启动 Solr。

现在你可以启动该 Solr 节点，此时升级后的 Solr 节点会重新加入到 SolrCloud 集群中，此时你需要查看 solr.log 日志信息，确认在该 Solr 节点启动之后是否有抛出异常。solr.log 日志文件存放在 solr.in.sh 脚本的 SOLR_LOGS_DIR 参数中进行配置。

第五步：运行集群健康检测。

在继续升级下一个 Solr 节点之前，你需要在刚刚升级完的 Solr 节点上执行健康检测，比如新升级的节点上有个名称为“myCollection”的 Collection，那么你可以运行下面的命令执行健康检测：

```
solr healthcheck -c myCollection -z ZK_HOST
```

查看是否有哪个 Replica 报告异常，最后你需要在你 SolrCloud 集群中的其他 Solr 节点上重复上述五个步骤。

16.2 Solr 中的伪域

在 Solr 中内置了很多“伪域”，你可以直接在查询参数通过 fl 参数进行指定。下面会一一介绍内置的每个伪域如何使用。

(1) [value]

```
q=*&fl=id,greeting:[value v='hello']
```

上面的示例中 greeting 表示一个自定义别名，相当于伪域的名称，v 属性表示伪域的域值。此外你还可以为 [value] 这个伪域指定数据类型，比如 [value v=42 t=int]，t 属性可选值有：int,float,double,date，注意：暂时不支持 long、boolean 类型。

(2) [shard]

```
q=*&fl=id,shard:[shard]
```

[shard] 伪域会返回当前被查询的 Collection 上的所有 Replica 节点的访问 URL。

(3) [docid]

这个伪域用于返回查询匹配的每个索引文档的内部 Lucene 的 DocumentID 值，一般用于调试。

(4) [elevated] 和 [excluded]

这两个伪域用于 Query Elevation 查询组件，即你需要事先在 solrconfig.xml 中配置 Query Elevation 查询组件。

[elevated] 伪域表示当前 Document 是否是推荐的。

[excluded] 伪域表示当前 Document 是否是被 Query Elevation 查询组件屏蔽的，前提是请求参数中 markExcludes=true。

这两个伪域使用示例如下：

```
fl=id,[elevated],[excluded]&excludeIds=GB18030TEST&elevateIds=6H500F0&markExcludes=true
```

(5) [subquery]

顾名思义，它是用来实现类似 SQL 里的 *where id in (select...)* 子查询的，举个例子说明，假如你有这样几个 Document：

```
{id: "D1", type_ss:"DEPT", name_ss:"Dept 1"}
{id: "S1", type_ss:"EMP", dept_ss:"D1", name_ss:"Emp 1 Dept 1"}
```

`type_ss=DEPT` 表示部门，`type_ss=EMP` 表示员工，`dept_ss` 表示部门 ID，与 `id` 域关联。此时如果想要返回所有部门同时返回该部门下的员工，你可以这样查询：

```
?fl=id,name_ss,emp:[subquery]&emp.rows=1&emp.fl=id,name_ss
&emp.q={!term f=dept_ss v=$row.id}&emp.fq=type_ss:"EMP"
&indent=on&q=*:*&fq=type_ss:"DEPT"&wt=json&expandMacros=true
```

`emp:[subquery]` 表示定义一个子查询，且 `[subquery]` 前面必须定义别名，然后你可以通过别名 `.q` 的方式来定义子查询的表达式，其中可以通过 `$row.id` 来进行外键关联，同理还有别名 `.fq`，别名 `.rows`。此外 `[subquery]` 伪域还支持跨 Core 查询，比如 `[subquery fromIndex=departments]`，这里的 `fromIndex` 表示其他 Core 的名称，但是 `fromIndex` 支持非 SolrCloud 模式，在 SolrCloud 模式下，你可以这样实现跨 Collection 的 `[subquery]`：

```
q=*:*&fl=*,foo:[subquery]&foo.q=cloud&foo.collection=departments
```



注意 `[subquery]` 伪域是 Solr6.1.0 中新添加的功能特性，它是 `[child]` 伪域的功能增强版。

此外，还有我们在其他章节介绍过的 `[explain]`、`[child]`、`[geo]` 等伪域。Solr 的伪域其实都是 `DocTransformer` 的子类。如果你想要为每个 Document 都添加一个自定义的伪域，那么你可以继承 `DocTransformer` 抽象类，重写其 `transform` 方法，通过 `transform` 方法中的 `SolrDocument` 入参你可以获取到当前 Document 的每个域值、域名称，通过 `setField` 或者 `addField` 方法往 Document 里再追加一些域，这里追加的域仅仅只是为了返回给前端用户显式，并不会写入索引。比如根据 Document 中的某个域的值经过计算进行求和求平均数，然后将计算值作为一个新域添加到 Document。比如还可以在 `transform` 方法内部根据 Document 中的某个域值从外部系统去加载数据，例如 Redis、MongoDB、MySQL，然后将从外部加载的数据作为伪域添加到 Document 中。但是你需要保证你在 `transform` 方法内部执行的操作开销不会很大，假如你的 Solr 主查询匹配了 10 万个 Document，那么意味着这 10 万 Document 都需要执行一次你在 `transform` 方法内部定义的操作，比如你在 `transform` 方法内部通过 JDBC 连接数据库，那么意味着你需要循环连接数据库 10 万，这是很恐怖的操作。

作。因此需要时刻清楚你在 transform 方法内部定义的操作会带来什么影响。

然后需要继承 TransformerFactory，在自定义的 TransformerFactory 中创建自定义的 DocTransformer 实现类，最后需要在 solrconfig.xml 中注册你自定义的 TransformerFactory，配置示例如下：

```
<transformer name="myTransform" class="xx.xx.XXXXXXFactory" ></transformer>
```

这里的 name 属性用于定义你的伪域名称，比如上面的 name="myTransform" 即表示你可以在 fl 参数中这样使用它：fl=[myTransform]，class 属性用于定义你自定义的 TransformerFactory 实现类的完整包路径。

16.3 Solr 多语种索引支持

在 Solr 中实现多语种的索引创建与查询，主要有以下几种实现方式：

- 为每一种语言创建一个单独的域进行存储。
- 为每一种语言创建一个 core，每个 Core 的 schema.xml 包含相同的域名称，仅仅只是分词处理不同。
- 自定义一个域类型能够支持对多语种文本进行索引与查询。

对于第一种实现方式，你只需要在 schema.xml 中针对每一种语言定义一个 <field> 并且每个 field 采用不同的 fieldType 进行分词处理，配置示例如下所示：

```
<field name="content_english" type="text_english" indexed="true"
stored="true" />
<field name="content_spanish" type="text_spanish" indexed="true"
stored="true" />
```

对于西班牙语，Solr 内置了 SpanishLightStemFilterFactory 可以用于对西班牙语进行分词处理，其他不支持的语种你可能需要自己自定义分词器来实现。然后添加索引文档阶段需要分别为每个语种对应的域设置域值，在查询阶段你需要跨多个域进行查询，比如 qf=content_spanish content_english。

但是这种方式存在一个弊端，如果需要支持的语种很多，比如几十种，那么意味着你需要在 schema.xml 中相应定义几十种 field 和 fieldType，同时你在添加索引文档时需要分别为每个语种设置域值，在 Solr 查询阶段需要跨几十个域进行查询，我们知道在多个域执行查询的性能要比在一个域上执行查询要慢得多。因此，这种方式只适合于你只需要支持 2~3 种语种的情况下使用。

为了解决以上提到的多个域上执行查询性能很慢，并且需要维护 N 个域等问题，你可以为每个语种单独创建一个 Core，这样你的每个 Core 的 schema.xml 定义基本一致，并且查询时你只需要并行请求多个 Core 在单个域上执行查询，查询性能得以提升。如果使用了 SolrCloud，那么你可以按照语种将索引划分为 N 个 Shard，N 取决于想要支持的语种个数。

然后你在查询时只需要指定 `shards` 参数来明确指定查询哪些 `Shard`。

第二种方式虽然能够解决问题，但是你需要管理多个 `Core` 或者管理 `SolrCloud` 集群，增加了索引和查询的复杂度，同时还需要额外投入成本维护 `SolrCloud` 集群。而且你必须手动对索引数据进行划分 `Shard`。

此时需要采用第三种方式来实现，通过这种方式拥有以下好处：

- 在单个域上支持 N 个语种，理论上 N 没有限制。
- 你只需要建立一个 `Core` 或者 `Collection`。
- 支持 `Solr` 中的任意 `Query Parser`，因此它仅仅只是在单个域上执行查询。
- 支持真正的多语种查询，并且支持多语种混杂在一起的短语查询。
- 不需要为每个语种数据进行双倍存储，减小了索引体积，从而使得查询性能提升。
- 同时还支持语种自动识别（`Solr` 中自带 `langid` 语言识别功能）。

具体实现代码请查阅随书源码中的第 16 章。想要使用它，你首先需要将其打包成 `jar` 包，然后你需要在 `schema.xml` 中配置 `field` 和 `fieldType`，以下是一个配置示例：

```
<fieldType name="multiText"
  class="com.yida.solr.book.examples.ch16.multilanguage.MultiTextField"
  sortMissingLast="true"
  defaultFieldType="text_general"
  fieldMappings="en:text_english,
    es:text_spanish,fr:text_french"/>
<field name="content" type="multiText" indexed="true" stored="true"
  multiValued="true"/>
<field name="language" type="string" indexed="true" stored="true" />
```

上面的 `fieldMappings` 属性用于配置不同语种对应的域类型之间映射，不同语种的检测是由 `Solr` 的 `langid` 功能提供，想要使用 `Solr` 的 `langid` 功能你需要额外导入 `langdetect-version.jar`、`jsonic-version.jar`、`solr-langid-version.jar`。`Solr` 中的 `langid` 自动识别语种采用的语种代号如下所示：

```
|af/Afrikaans| |ar/Arabic| |bg/Bulgarian| |bn/Bengali| |cs/Czech| |da/
Danish| |de/German| |el/Greek| |en/English| |es/Spanish| |et/Estonian| |fa/
Persian| |fi/Finnish| |fr/French| |gu/Gujarati| |he/Hebrew| |hi/Hindi| |hr/
Croatian| |hu/Hungarian| |id/Indonesian| |it/Italian| |ja/Japanese| |kn/Kannada|
|ko/Korean| |lt/Lithuanian| |lv/Latvian| |mk/Macedonian| |ml/Malayalam| |mr/
Marathi| |ne/Nepali| |nl/Dutch| |no/Norwegian| |pa/Punjabi| |pl/Polish| |pt/
Portuguese| |ro/Romanian| |ru/Russian| |sk/Slovak| |sl/Slovene| |so/Somali| |sq/
Albanian| |sv/Swedish| |sw/Swahili| |ta/Tamil| |te/Telugu| |th/Thai| |tl/Tagalog|
|tr/Turkish| |uk/Ukrainian| |ur/Urdu| |vi/Vietnamese| |zh-cn/Simplified Chinese|
|zh-tw/Traditional Chinese|
```

然后你需要在 `solrconfig.xml` 中配置 `MultiTextFieldLanguageIdentifierUpdateProcessorFactory`，示例如下：

```
<updateRequestProcessorChain name="multi-langid">
```

```

<processor class="com.yida.solr.book.examples.ch16
.multilanguage.MultiTextFieldLanguageIdentifierUpdateProcessorFactory">
<lst name="invariants">
<str name="langid.fl">title,content</str>
<str name="langid.langField">language</str>
<str name="langid.whitelist">en,es,fr</str>
<str name="langid.fallback">en</str>
</lst>
<lst name="defaults">
<str name="mtf-langid.prependFields">content</str>
<str name="mtf-langid.prependGranularity">fieldValue</str>
<str name="mtf-langid.hidePrependedLangs">false</str>
</lst>
</processor>
<processor class="solr.LogUpdateProcessorFactory" />
<processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
<requestHandler name="/update" class="solr.UpdateRequestHandler">
<lst name="invariants">
<str name="update.chain">multi-langid</str>
</lst>
</requestHandler>

```

langid.fl 表示需要对哪些域进行自动语种识别, langid.langField 表示自动识别出来的语种存储到哪个域上, langid.whitelist 用于限制 solr 自动识别出来的语种必须在此参数限定的范围内, 否则不予以接受。langid.fallback 表示当 Solr 检测不出来语种或者不在 langid.whitelist 参数限定范围内, 此时会采用 langid.fallback 作为默认值。

然后你就创建索引数据, 紧接着在查询时, 可以指定在哪些语种上进行查询, 示例如下 (完整示例请查阅随书源码):

```

http://localhost:8080/solr/multilanguage/select?
q=en,es,fr|abandon& df=content&fl=title,content,language

```

16.4 Solr 中自定义 Redis 缓存

在 Solr 中默认内置了 3 种缓存实现, 如果想要在 Solr 中使用更高效的缓存, 那么你可以考虑使用 Redis。但是你需要继承 SolrCacheBase 抽象类同时实现 SolrCache 接口。具体实现源码请查阅随书源码的第 16 章 cache 包下的实现。下面简单介绍下应该如何使用随书源码中自定义的 SolrRedisCache。

首先需要将其打包成 jar 并导入 Solr 保证 Solr 能够正确加载它, 此外 SolrRedisCache 需要额外依赖两个 jar 包: jedis 和 spymemcached。具体请查阅随书源码中 pom.xml 的定义。然后可以将 Solr 缓存的实现类替换为自定义的 SolrRedisCache, 配置示例如下所示:

```

<filterCache name="redisCache"

```

```
class="com.yida.solr.book.examples.ch16.cache.SolrRedisCache"
host="linux.yida.com"
port="6379"
size="4096"
initialSize="1024"
autowarmCount="1024"/>
```

16.5 Solr 如何开启 HTTPS

单机版 Solr 和 SolrCloud 都支持使用 SSL 进行加密通信。本章节将详细描述如何在 Jetty 容器下使用自签名证书来启用 SSL。

先说说单机版 Solr 环境下，首先需要使用 JDK bin 目录下自带的 keytool 工具生成自签名证书和 key。请在 /opt/solr-6.2.1/server/etc 目录下执行如下命令：

```
keytool -genkeypair -alias solr-ssl -keyalg RSA -keysize 2048 -keypass secret
-storepass secret -validity 9999 -keystore solr-ssl.keystore.jks -ext SAN=DNS:localhost,IP:192.168.1.3,IP:127.0.0.1 -dname "CN=localhost, OU=Organizational Unit, O=Organization, L=Location, ST=State, C=Country"
```

-alias 表示密钥的别名，-keypass 表示你的密钥口令，-keyalg 用于指定密钥的生成算法，-storepass 表示密钥库的密码，-keystore 用于指定生成的密钥库文件保存路径，如果不指定路径，那么默认会保证在当前路径下。-ext SAN 用于设置 DNS 名称和主机验证时允许的 IP 或域名。-validity 用于指定创建的证书有效期是多少天。其他参数信息无关紧要随便填。

然后你需要将证书转换成 cURL 能理解的 PEM 格式，先使用 keytool 将 JKS keystore (密钥库) 转换成 PKCS12 格式，执行命令如下：

```
keytool -importkeystore -srckeystore solr-ssl.keystore.jks -destkeystore solr-ssl.keystore.p12 -srcstoretype jks -deststoretype pkcs12
```

-srckeystore 参数需要与上一个命令中的 -keystore 参数保持一致，默认是相对当前路径。同理 -destkeystore 参数用于指定生成的 p12 文件保存路径，默认保存在当前路径下。

紧接着，使用 openssl 命令将 PKCS12 格式的密钥库转换成 PEM 格式：

```
openssl pkcs12 -in solr-ssl.keystore.p12 -out solr-ssl.pem
```

但是你需要提前安装 OpenSSL。OpenSSL 安装包下载地址如下：

<https://www.openssl.org/source/old/>

OpenSSL 安装步骤大致如下：

```
tar -xzf openssl-1.1.0.tar.gz -C /opt/modules // 解压 OpenSSL 安装包
cd /opt/modules/openssl-1.1.0
./config --prefix=/opt/modules/openssl
```


```
./config -t
make           // 编译 OpenSSL, 你需要确保你已经安装了 gcc 编译器
make install   // 安装 OpenSSL
```

如果想要在 OS X Yosemite (一款苹果操作系统) 上使用 cURL, 那么你还需要创建一个 PEM 格式的只包含证书的版本, 执行命令如下:

```
openssl pkcs12 -nokeys -in solr-ssl.keystore.p12 -out solr-ssl.cacert.pem
```

然后你需要在 `/etc/default/solr.in.sh` 脚本中激活有关 SSL 相关的配置, 示例如下:

```
SOLR_SSL_KEY_STORE=etc/solr-ssl.keystore.jks
SOLR_SSL_KEY_STORE_PASSWORD=secret
SOLR_SSL_TRUST_STORE=etc/solr-ssl.keystore.jks
SOLR_SSL_TRUST_STORE_PASSWORD=secret
SOLR_SSL_NEED_CLIENT_AUTH=false
SOLR_SSL_WANT_CLIENT_AUTH=false
```

 **注意** SOLR_SSL_NEED_CLIENT_AUTH 和 SOLR_SSL_WANT_CLIENT_AUTH 是互斥的, 请不要将两者同时配置为 `true`。SOLR_SSL_NEED_CLIENT_AUTH 表示是否需要验证客户端证书, 如果设置为 `true`, 则表示强制双向 SSL 验证, 否则即表示单向 SSL 验证。SOLR_SSL_WANT_CLIENT_AUTH 表示可以验证客户端证书, 但如果客户端没有有效证书, 也不强制验证。

此时 Solr 客户端访问 Solr 需要带上如下几个系统参数, 当然你也可以采用 `-Dkey=val` 的格式在 Solr 启动参数中添加它们:

```
System.setProperty("javax.net.ssl.keyStore", "/path/to/solr-ssl.keystore.jks");
System.setProperty("javax.net.ssl.keyStorePassword", "secret");
System.setProperty("javax.net.ssl.trustStore", "/path/to/solr-ssl.keystore.jks");
System.setProperty("javax.net.ssl.trustStorePassword", "secret");
```

下面简单介绍下 Tomcat 容器下如何为 Solr 开启 HTTPS, 以下是具体操作步骤:

```
# 使用 JDK 自带的 keytool 工具生成证书
keytool -genkey -alias tomcat -keyalg RSA -keystore /opt/tomcat.keystore
# 生成自签名证书
keytool -selfcert -alias tomcat -keystore /opt/tomcat.keystore
# 导出证书
keytool -export -alias tomcat -keystore /opt/tomcat.keystore -storepass 123456 -rfc -file /opt/tomcat.cer
# 将证书导入到客户端的 JVM 中, 即信任此证书
keytool -import -keystore /opt/jdk1.7.0_79/jre/lib/security/cacerts -file /opt/tomcat.cer -alias tomcat
```

在 Tomcat 的 server.xml 中应用该证书:

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11Protocol"
    maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
    clientAuth="false" sslProtocol="TLS" keystorefile="/opt/tomcat.keystore" keystorepass=
    "123456"/>
```

如果想要强制 Solr 必须工作在 SSL 下, 那你可以简单地在 Solr 的 web.xml (在 /opt/modules/apache-tomcat-8.5.6/webapps/solr/WEB-INF 目录下) 中添加如下配置:

```
<security-constraint>
<web-resource-collection>
<web-resource-name>solr</web-resource-name>
<url-pattern>/*</url-pattern>
</web-resource-collection>
<user-data-constraint>
<!-- 设置为 NONE 即表示禁用 SSL -->
<transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
</security-constraint>
```

16.6 Solr 安全认证

Solr 拥有一套安全框架用于支持用户安全认证与授权, 它能够验证用户的身份以及严格控制用户对 Solr 集群中资源的访问权限。Solr 包含了一些安全认证与授权相关插件, 这些插件大部分都只能应用于 SolrCloud 模式下。在 SolrCloud 模式下, 想要使用这些插件, 首先需要将 security.json 文件上传到 ZooKeeper。如果该插件还支持单机模式, 你需要通过设置一个系统参数 -DauthenticationPlugin= 插件实现类来帮你在 ZooKeeper 中创建并管理 security.json。下面是 Solr 中支持的安全认证相关插件:

❑ Solr 自带的基础安全认证: 只适用于 SolrCloud。

❑ Kerberos 安全认证: 支持 SolrCloud 模式与单机模式。

❑ 基于规则的安全认证: 只适用于 SolrCloud。

❑ 基于 PKI 的安全认证: 只适用于 SolrCloud。

16.6.1 基础安全认证插件

要想使用 Solr 中的基础安全认证插件, 首先需要将 security.json 文件上传到 Zookeeper 中。请自己手动创建 security.json 文件, 并按照如下配置示例进行配置:

```
{
  "authentication":{
    "class":"solr.BasicAuthPlugin",
    "credentials":{
      "solr":"IV0EHq1OnNrj6gvRCwvFwTrZ1+z1oBbnQdiVC3otuq0= Ndd7LKvVBAAZI
```



```

FOQAVilekCfAJXr1GGfLtRUXhgrF8c="
    }
},
"authorization":{
    "class":"solr.RuleBasedAuthorizationPlugin",
    "permissions":[
        {"name":"security-edit","role":"admin"},
        {"name":"restrict_collections","role":"admin"},
        {"name":"restrict_select","role":"admin"},
        {"name":"restrict_update","role":"admin"},
        {"name":"read","role":"admin"}
    ],
    "user-role":{"solr":"admin"}
}
}

```

以上的配置中 authentication 部分用于设置账号密码，其中 class 表示设置安全认证插件的实现类，除了 BasicAuthPlugin，可选的还有：KerberosPlugin。credentials 部分用于配置账号密码，其中 solr 表示自定义的账号，后面的一串为加密后的密码字符串，采用的是 SHA-256 加密方式，实现代码如下所示：

```

public static void main(String[] args) {
    // 原始明文密码
    String pwd = "solr";
    pwdEncrypt(pwd);
}
// 对密码进行 SHA-256 密码加密
public static String pwdEncrypt(String pwd) {
    System.out.println(" 加密前密码: " + pwd);
    final Random r = new SecureRandom();
    byte[] salt = new byte[32];
    r.nextBytes(salt);
    String saltBase64 = Base64.encodeBase64String(salt);
    String val = sha256(pwd, saltBase64) + " " + saltBase64;
    System.out.println(" 加密后密码: " + val);
    return val;
}
public static String sha256(String password, String saltKey) {
    MessageDigest digest;
    try {
        digest = MessageDigest.getInstance("SHA-256");
    } catch (NoSuchAlgorithmException e) {
        return null;
    }
    if (saltKey != null) {
        digest.reset();
        digest.update(Base64.decodeBase64(saltKey));
    }
    byte[] btPass = digest.digest(password.getBytes(StandardCharsets.UTF_8));
}

```

```

    digest.reset();
    btPass = digest.digest(btPass);
    return Base64.encodeBase64String(btPass);
}

```

authorization 部分用于配置角色权限授权, class 表示授权插件实现类, user-role 部分用于为帐号分配角色, 前为帐号后为角色名称, permissions 部分用于为角色分配权限, role 属性表示角色名称, name 属性表示该角色对应的权限名称。

执行如下命令将 security.json 文件上传到 Zookeeper:

```

/opt/solr-6.2.1/server/scripts/cloud-scripts
zkcli.sh -z linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181
-cmd putfile /security.json /opt/solr6/security.json

```

执行如下命令重启集群各个节点:

```

solr restart -c -h linux.yida01.com -p 8983 -z linux.yida01.com:2181,linux.
yida02.com:2181,linux.yida03.com:2181 -s /opt/solr_home

```

访问如下链接验证安全认证插件配置是否生效:

<http://linux.yida01.com:8983/solr/admin/authentication>

如果提示你: "errorMessages":["No authentication configured"]}, 则表明配置没有成功, 请检查你的 security.json 配置。如果你能看到类似如下的提示信息, 则表明配置成功了:

```

"authentication.enabled":true,
  "authentication":{
    "class":"solr.BasicAuthPlugin",
    "credentials":{"solr":{"IV0EHq1OnNrj6gvRCwvFwTrZ1+z1oBbnQdiVC3otuq0= Nd
d7LKvVBAAzIF0QAVilekCfAJXr1GGfLrUXhgrF8c="}}}

```

此时刷新你的 Solr Web 后台管理界面, 会弹出账号密码输入框, 只有正确输入账号密码才能访问 Solr。上面的 security.json 中我配置的账号密码是 solr: SolrRocks。密码请使用上面提供的 pwdEncrypt 方法生成。

下面介绍 security.json 中 authorization (授权) 部分的 permissions 属性里, 你可以配置的权限:

- ☐ security-edit: 可以修改 security.json 文件。
- ☐ security-read: 只能查看 security.json 文件。
- ☐ schema-edit: 拥有修改所有 Collection 的 schema.xml 文件的权限。
- ☐ schema-read: 拥有查看所有 Collection 的 schema.xml 文件的权限。
- ☐ config-edit: 拥有修改所有 Collection 的 solrconfig.xml 和请求参数的权限。
- ☐ config-read: 拥有查看所有 Collection 的 solrconfig.xml 和请求参数的权限。
- ☐ core-admin-read: 拥有 Core Admin API 的读权限。
- ☐ core-admin-edit: 拥有 Core Admin API 的写权限, 此权限可以改变系统状态。

- ❑ collection-admin-edit: 拥有对 Collection 的一切管理权限, 比如添加 / 删除 Collection, 添加 / 删除 Shard、Replica、分割 Sard 等。
- ❑ collection-admin-read: 拥有对 Collection 配置的查看权限, 比如查看集群状态, 执行的 action 有: OVERSEERSTATUS、CLUSTERSTATUS、REQUESTSTATUS。
- ❑ update: 拥有对所有 Collection 执行 Update 操作的权限, 比如发送索引文档到 Solr Server 创建索引。
- ❑ read: 拥有对所有 Collection 执行读操作 (即 Solr 查询操作) 的权根, 比如 /select、/get、/browse、/tvrh、/terms、/clustering、/elevate、/export、/spell、/clustering、/sql。
- ❑ all: 拥有操作 Solr 的所有权限。

16.6.2 Solr 中的 Authorization API

除了可以编辑 security.json 文件来对用户进行授权之外, 你还可以通过 Solr 提供的 Authorization API 来动态地添加用户、角色, 以及为用户角色授权。

Authorization API 接口 URL 为: /admin/authorization, 支持 4 种命令来控制权限管理:

- ❑ set-permission: 创建一个权限, 重写已经存在的权限定义, 或者为角色分配一个已经提前定义好的权限。
- ❑ update-permission: 更新已经存在的权限定义的一些属性。
- ❑ delete-permission: 删除一个权限定义。
- ❑ set-user-role: 为用户分配角色。

在自定义权限时你可以使用表 16-1 中列举的属性。

表 16-1 Authorization API 自定义权限支持的参数表

参 数	描 述
name	表示权限的名称, 当你需要预定义权限时才需要指定此参数
collection	表示你定义的权限需要应用到哪个 Collection 上
path	path 表示 Request Handler 在 solrconfig.xml 中定义的名称, 比如 /select、/update, 此参数支持通配符, 比如 /update/*
method	用于限制能够客户端发送的 HTTP 请求使用的 Method, 可选值有 GET,POST,PUT,DELETE,HEAD
params	<p>用于限制可以执行的 Collection API 的 action, 比如:</p> <pre>"params": { "action": [LIST, CLUSTERSTATUS] }</pre> <p>同时 action 参数值还支持正则表达式, 比如:</p> <pre>"params": { "action": ["REGEX:(?i)LIST","REGEX:(?i)CLUSTERSTATUS"] }</pre>
before	用于定义权限的顺序, 表示在 security.json 新权限应该排在前面, 顺序值在权限创建时自动按序分配
role	表示定义的权限应该分配给哪个角色。此参数值可以使用通配符

下面列举几个 SolrAuthorization API 的示例:

1) 创建一个权限:

```
http://localhost:8983/solr/admin/authorization
```

```
"set-permission": {
  "collection": null,
  "path": "/admin/collections",
  "params": {"action": [LIST, CREATE]},
  "before": 3, "role": "admin"
}
```

2) 更新一个权限 (将权限分配给指定的角色以及更新权限的索引值):

```
{ "update-permission": {
  "index": 3, "role": ["admin", "dev"]}
}
```

3) 删除一个权限:

```
{ "delete-permission": 3 }
```

4) 为用户分配角色:

```
{ "set-user-role": {
  "solr": ["admin", "dev"], "harry": null
}
```

solr 表示一个用户名称, 中括号内表示分配给用户的角色, 如果你想要收回分配给某个用户的角色, 请将角色属性值设置为 null。

当你为 Solr 开启了安全认证, 此时你的 SolrJ 客户端访问 Solr Server 的代码需要进行修改, 在随书源码中提供了一个完整示例, 为了节省篇幅这里就不贴实现代码了, 请大家自行阅读随书源码。

16.7 SolrCloud 模式下使用 Canal 增量更新索引

单机模式下一般采用 Solr 内置的 DIH 功能, 然后不断轮询的方式可以实现 Solr 索引数据的增量更新。然而, 对于大规模的 SolrCloud 集群而言, 这显然不可取, 此时你可以考虑借助 Alibaba 开源的 Canal 增量数据订阅 & 消费中间件, 它支持增量的解析关系型数据库的 binary log 日志。Canal 项目的 Github 地址为: <https://github.com/alibaba/canal>。Canal 官方提供的学习文档: <http://alibaba.github.io/canal/>。

首先你需要安装 MySQL 数据库, 下面是 MySQL 数据库的详细安装过程, 这里是采用源码编译的方式来安装 MySQL 数据库。

首先安装编译 MySQL 所依赖的包:

```
yum -y install make gcc-c++ cmake bison-devel ncurses-devel perl perl-devel
```

MySQL 安装包下载地址:

<http://mirrors.sohu.com/mysql/>

解压 MySQL 安装包:

```
tar -xzf mysql-5.6.14.tar.gz -C /opt/modules/
cd /opt/modules/mysql-5.6.14/
```

编译 MySQL:

```
cmake \
-DMAKE_INSTALL_PREFIX=/usr/local/mysql \
-DMYSQL_DATADIR=/usr/local/mysql/data \
-DSYSCONFDIR=/etc \
-DWITH_MYISAM_STORAGE_ENGINE=1 \
-DWITH_INNOBASE_STORAGE_ENGINE=1 \
-DWITH_MEMORY_STORAGE_ENGINE=1 \
-DWITH_READLINE=1 \
-DMYSQL_UNIX_ADDR=/var/lib/mysql/mysql.sock \
-DMYSQL_TCP_PORT=3306 \
-DENABLED_LOCAL_INFILE=1 \
-DWITH_PARTITION_STORAGE_ENGINE=1 \
-DEXTRA_CHARSETS=all \
-DDEFAULT_CHARSET=utf8 \
-DDEFAULT_COLLATION=utf8_general_ci
```

安装 MySQL:

```
make && make install
```

开始进行 MySQL 数据库初始化配置, 首先创建一个 mysql 系统用户, 专门用于 MySQL 系统服务的操作控制:

```
groupadd mysql
useradd -g mysql mysql
```

修改 /usr/local/mysql 目录所有者为 mysql 用户:

```
chown -R mysql:mysql /usr/local/mysql
```

为了防止系统默认的 /etc/my.cnf 配置干扰 MySQL 数据库安装, 请将其重命名, 如下所示:

```
mv /etc/my.cnf /etc/my.cnf_
```

开始安装 MySQL 自带的数据库表:

```
cd /usr/local/mysql
scripts/mysql_install_db --basedir=/usr/local/mysql --datadir=/usr/local/
mysql/data --user=mysql
```

使用 vi 命令编辑 /usr/local/mysql/my.cnf 配置文件:

```

[mysqld]
init_connect='SET collation_connection = utf8_unicode_ci'
init_connect='SET NAMES utf8'
character-set-server=utf8
collation-server=utf8_unicode_ci
skip-character-set-client-handshake
datadir=/usr/local/mysql/data
socket=/var/lib/mysql/mysql.sock
sql_mode=NO_ENGINE_SUBSTITUTION,STRICT_TRANS_TABLES
[mysql]
default-character-set=utf8
socket=/var/lib/mysql/mysql.sock
[mysql.server]
user=mysql
basedir=/usr/local/mysql
[safe_mysqld]
err-log=/usr/local/mysql/mysqld.log
pid-file=/usr/local/mysql/mysqld.pid
[client]
default-character-set=utf8
socket=/var/lib/mysql/mysql.sock

```

添加 MySQL 系统服务，复制服务脚本到 init.d 目录，并设置随操作系统开机启动：

```

cp support-files/mysql.server /etc/init.d/mysql
chkconfig mysql on

```

启动 MySQL 系统服务：

```
service mysql start
```

设置 MySQL 环境变量，便于我们能够在任意路径下执行 mysql 命令，示例如下所示：

```

MYSQL_HOME=/usr/local/mysql
PATH=$PATH:$MYSQL_HOME/bin

```

最后 source/etc/profile 使 MySQL 环境变量设置立即生效。

登录 MySQL（默认安装完成之后 root 用户的密码为空），修改 MySQL 默认 root 用户的密码，具体操作如下所示：

```

mysql -uroot
use mysql
update user set password=PASSWORD('123') where user='root';
flush privileges;

```

现在开始搭建 MySQL 主从复制架构。这里假定 linux.yida01.com 这台主机作为 MySQL Master 节点，在 Master 数据库上创建 master 账号，同时赋予权限：

```

create user master identified by '123';
grant all privileges on *.* to master@%' identified by '123' with grant option;
flush privileges;

```


修改 /usr/local/mysql/my.cnf 配置文件

```
[mysqld]
log-bin=mysql-bin
binlog-format=ROW
server-id=1
```

每台 MySQL 节点都需要配置, 且 server-id 参数值必须全局唯一, 修改之后 MySQL 服务需要重启才能立即生效。

查看 Master 的状态, 获取 master_log_file 文件名和 Position 值。Slave 中配置 Master 时需要这两个参数:

```
show master status;
```

配置 Slave 节点的 master, 具体操作如下所示:

```
mysql -uroot -p123
use mysql
change master to master_host='linux.yida01.com',master_user='master',master_
password='123',master_log_file='mysql-bin.000001',master_log_pos=120;
start slave; // 启动 Slave
show slave status; // 查看 Slave 的状态
```

下面开始搭建 Canal 运行环境, 首先请在 MySQL Master 节点上创建一个账号和密码都为 canal 的 MySQL 用户, 并赋予权限, 如下所示:

```
CREATE USER canal IDENTIFIED BY 'canal';
GRANT SELECT, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'canal'@'%'
identified by 'canal';
FLUSH PRIVILEGES;
```

然后你需要从 Canal 的 Github 上下载其源码, 并导入到 IDEA 或 Eclipse 中, 对其进行源码编译, 执行命令如下所示:

```
mvn clean install -Dmaven.test.skip -Denv=release
```

在项目根目录下的 target 目录下得到一个 canal.deployer-version.tar.gz 的 Canal 部署包。对其进行解压, 执行命令如下所示:

```
mkdir -p /opt/modules/canal
tar -zxf canal.deployer-1.0.23-SNAPSHOT.tar.gz -C /opt/modules/canal/
```

对 canal 的 conf 目录下的 canal.properties 属性文件进行配置修改, 示例如下:

```
canal.ip= linux.yida01.com
canal.port= 11111
canal.zkServers= linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.
com:2181
```

对 conf/example/instance.properties 属性文件进行配置修改, 示例如下:

```

## mysql serverId
canal.instance.mysql.slaveId = 1234
# position info
canal.instance.master.address = linux.yida01.com:3306
canal.instance.master.journal.name = mysql-bin.000002
canal.instance.master.position = 1252
# username/password
canal.instance.dbUsername = canal
canal.instance.dbPassword = canal
canal.instance.defaultDatabaseName = canal
canal.instance.connectionCharset = UTF-8

```

上面配置涉及的一些配置参数的详细解释请参见表 16-2。

表 16-2 MySQL 集群示例参数表

配置参数	描 述
canal.instance.mysql.slaveId	当前 MySQL 集群中 Slave 节点的 ID，需要保证全局唯一，这里 canal 将自身伪装成一个 slave 节点，因此需要配置 slaveId
canal.instance.master.address	MySQL 集群中 Master 节点的 ip（或域名）与端口号
canal.instance.master.journal.name	MySQL 集群中 Master 节点的 binlog 文件名称，可以通过 show master status 查看
canal.instance.master.position	MySQL 集群中 Master 节点的 binlog 文件的起始偏移量，可以通过 show master status 查看
canal.instance.master.timestamp	MySQL 集群中 Master 节点的 binlog 文件的起始时间戳，可以通过 show master status 查看
canal.instance.dbUsername	Canal 连接 Master 数据库使用的账号
canal.instance.dbPassword	Canal 连接 Master 数据库使用的密码
canal.instance.defaultDatabaseName	Canal 默认连接 Master 节点的哪个数据库
canal.instance.connectionCharset	Canal 连接 Master 时使用的字符集编码

配置完成之后，切到 canal 的 bin 目录下执行 start.up.sh 脚本来启动 Canal 实例，执行命令如下所示：

```
sh bin/startup.sh
```

通过 jps 命令来查看是否已经存在一个名称为 CanalLauncher 的系统进程，如果存在则表明 Canal 实例已经启动成功了。Canal 的日志文件默认存储在 Canal 的 logs/canal 和 logs/example（example 为 Canali 的默认实例名称）目录下，启动过程中，如果出现任何异常请随时查阅 Canal 日志文件便于你排错。

将编译 Canal 源码后在 canal-master\client\target 目录下生成的 canal.client-version.jar 安装到本地 Maven 仓库，具体操作命令如下所示：

```

mvn install:install-file -DgroupId=com.alibaba.otter -DartifactId=canal.client
-Dversion=1.0.23-SNAPSHOT -Dpackaging=jar -Dfile=E:/zip/canal-master/client/

```

target/canal.client-1.0.23-SNAPSHOT.jar

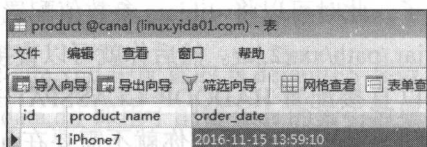
同理还有 canal.protocol 和 canal.common，然后你就可以在你的 Maven 项目的 pom.xml 中添加对 canal-client、canal.protocol 和 canal.common 这 3 个 Canal 模块的依赖，示例如下：

```
<canal.version>1.0.23-SNAPSHOT</canal.version>
<dependency>
<groupId>com.alibaba.otter</groupId>
<artifactId>canal.client</artifactId>
<version>${canal.version}</version>
</dependency>
<dependency>
<groupId>com.alibaba.otter</groupId>
<artifactId>canal.protocol</artifactId>
<version>${canal.version}</version>
</dependency>
<dependency>
<groupId>com.alibaba.otter</groupId>
<artifactId>canal.common</artifactId>
<version>${canal.version}</version>
</dependency>
```

将 <http://alibaba.github.io/canal/> 这里提供的 ClientSample 类示例代码复制到你的项目中运行测试，或者直接从随书源码中获取。Canal 客户端连接示例代码中比较关键的地方就是创建 CanalConnector 对象部分：

```
CanalConnector connector = CanalConnectors.newSingleConnector(
    new InetSocketAddress("linux.yida01.com",
        11111), "example", "canal", "canal");
```

newSingleConnector 方法的第一个参数用于指定 Canal 实例所在主机的 IP 或域名，第二个参数表示 Canal 实例监听的端口号，第 3 个参数表示 Canal 实例的名称，默认为 example，第 4 个参数表示 Canal 实例连接 Master 时使用的账号，第 5 个参数表示 Canal 实例连接 Master 时使用的密码。这些参数都需要与我们在 canal 的 conf 目录下的 properties 属性文件中的配置保持一致。然后你就可以运行该测试类，同时可以在数据库（即在 instance.properties 属性文件中配置的 canal.instance.defaultDatabaseName 参数值对应的数据库）中创建一张表，然后往该表中添加一条测试数据来触发数据库表数据变更，同时观察 IDEA 控制台打印信息。



id	product_name	order_date
1	iPhone7	2016-11-15 13:59:10

图 16-1 Canal 实例测试表

要与我们在 canal 的 conf 目录下的 properties 属性文件中的配置保持一致。然后你就可以运行该测试类，同时可以在数据库（即在 instance.properties 属性文件中配置的 canal.instance.defaultDatabaseName 参数值对应的数据库）中创建一张表，然后往该表中添加一条测试数据来触发数据库表数据变更，同时观察 IDEA 控制台打印信息。

```
=====> binlog[mysql-bin.000002:2352], name[canal.product], eventType : INSERT
id : 1      update=true
product_name : iPhone7      update=true
order_date : 2016-11-15 13:59:10      update=true
```

图 16-2 Canal 实例打印的增量数据信息

如图 16-2 所示, 获取到增量数据之后, 你可以根据 Canal 返回的事件类型来使用 Solr 原子更新对应的操作符比如 set、add、remove 来构建 SolrInputDocument, 具体如何构建请参见随书源码第 16 章的 canal 包下的 TestCanalClient 类中的 createDeltaSolrDocument 方法。最后你只要调用 SolrJ 的原子更新接口即可实现 Solr 索引数据的增量更新。

16.8 Solr 与 MapReduce 集成

为了提升 Solr 索引创建的性能, 你可以借助 MapReduce 这个分布式计算框架将 Solr 索引创建任务拆分成多个子任务然后在多台服务器上并行执行。MapReduce 任务一般分成 Map 和 Reduce 两个阶段, 由于我们创建 Solr 索引不需要合并过程, 因此我们只需要继承 org.apache.hadoop.mapreduce.Mapper 类, 重写其 map 方法, 在 map 方法中实现自己的业务逻辑。在随书源码的第 13 章中已经提供了一个读取 HDFS 上的文件创建 Solr 索引的示例, 此外随书源码的第 16 章中也提供了一个 CSVIndexMapper 实现, 能够支持对 HDFS 上的 CSV 格式文件创建 Solr 索引, 希望这两个示例能够给大家一些启发, 至于关系型数据库表里的数据, 你可以先将表里的数据导出为 CSV 格式, 然后根据本书提供的 CSVIndexMapper 来实现 Solr 索引创建。你只需要将 ch16.mapreduce 包下的类打包成 jar, 然后在 hadoop 服务器上执行即可, 执行命令如下所示:

```
hadoop jar xxxxxx.jar xxx.xx.SolrIndexLauncher -libjars /path/xxx1.jar,/path/xxx2.jar
```

上面的 -libjars 参数用于配置索引创建任务运行类 SolrIndexLauncher 依赖的第三方 jar 包, 多个 jar 包采用逗号分割。当依赖的 jar 包很多, 编写这个执行命令就显得不够方便了, 此时可以将 -libjars 参数值配置到系统环境变量中, 比如: export LIBJARS=/path/xxx1.jar,/path/xxx2.jar。然后你就可以直接以 -libjars \${LIBJARS} 方式来添加依赖 jar 包, 或者直接配置 HADOOP_CLASSPATH 环境变量, 如果你在 /etc/profile 中配置了 HADOOP_CLASSPATH, 那么你就不需要在 hadoop jar 命令中显式的指定 -libjars 参数, Hadoop 会自动从你配置的 HADOOP_CLASSPATH 系统变量下查找依赖的 jar 包, 配置示例如下所示:

```
export HADOOP_CLASSPATH=/path/xxx1.jar:/path/xxx2.jar
```



注意 上面的 HADOOP_CLASSPATH 系统变量值中, 多个 jar 包使用冒号字符进行分割而不是逗号。

关于 Solr 与 Mapreduce 集成更完整的集成方案推荐使用 Github 上的一个开源项目: hadoop-solr: <https://github.com/lucidworks/hadoop-solr>。它支持 Solr5.x 和 Hadoop2.x, 支持对 CSV、XML 格式文件创建索引, 支持对指定目录下的所有文件创建索引。不过此项目是

基于 Gradle 构建而不是 Maven，因此你还需要额外的学习 Gradle 的使用。限于篇幅有限，这里就留给大家自行学习研究了。

16.9 Solr 使用 HDFS 存储索引

Solr 支持将索引数据文件、配置文件以及事务日志文件存储在 HDFS 上，然后 Solr 直接读写 HDFS，而非本地文件系统。这里仅仅只是利用 HDFS 分布式文件系统来存储 Solr 相关文件，并没有利用 MapReduce 来创建索引。想要在 Solr 中使用 HDFS，必须使用 Hadoop2.x，并且你需要指示你的 Solr 使用 HdfsDirectoryFactory。这里有 3 种指定方式：

- ❑ 当你使用 bin/solr 脚本启动 Solr 时传递 JVM 参数，但是这种方式需要每次使用该脚本启动 Solr 时都重新传递。并且这种方式是全局生效，即会影响 Solr 中的所有 Collection。
- ❑ 修改 solr.in.sh，在其中配置 JVM 参数，这样当使用 bin/solr 脚本启动 Solr 时，Solr 会自动应用 solr.in.sh 中配置的参数，这样你就不需要每次都手动指定了。但是这种方式是全局生效，即会影响 Solr 中的所有 Collection。
- ❑ 在 solrconfig.xml 定义 HdfsDirectoryFactory。但是这种方式需要为每个 Collection 对应的 solrconfig.xml 进行配置。当你只希望部分 Collection 存储到 HDFS，而其他部分 Collection 存储到本地文件系统时，此种方式会比较合适。

对于 Solr 单机模式，你可以在启动 Solr 之前传递如下几个 JVM 参数：

```
bin/solr start -Dsolr.directoryFactory=HdfsDirectoryFactory
-Dsolr.lock.type=hdfs
-Dsolr.data.dir=hdfs://host:port/path
-Dsolr.updateLog=hdfs://host:port/path/tlog
```

其中 hdfs://host:port 部分需要与你在 hadoop 的 core-site.xml 中配置的 fs.defaultFS 参数值保持一致。后面的 /path 即 HDFS 上的文件路径。solr.data.dir 即用于配置 Solr 的索引数据目录，而 solr.updateLog 则用于配置 Solr 的事务日志文件存放目录。

如果使用的是 SolrCloud，那么在启动 Solr 之前你可以这样传递 JVM 参数：

```
bin/solr start -c -Dsolr.directoryFactory=HdfsDirectoryFactory
-Dsolr.lock.type=hdfs
-Dsolr.hdfs.home=hdfs://host:port/path
```

solr.hdfs.home 用于配置 Solr 在 HDFS 上的 SOLR_HOME 目录，只是为了区分普通的 solr.solr.home 参数，两者配置含义是基本相同的。

对于 SolrCloud 模式，你还可以在 solr.in.sh 配置文件配置 JVM 参数，而不用每次启动 Solr 会临时指定这些参数，配置示例如下所示：

```
# Set HDFS DirectoryFactory & Settings
```

```
-Dsolr.directoryFactory=HdfsDirectoryFactory \
-Dsolr.lock.type=hdfs \
-Dsolr.hdfs.home=hdfs://host:port/path \
```

第三种指定方式就是在 solrconfig.xml 中配置 HdfsDirectoryFactory，配置示例如下所示：

```
<directoryFactory name="DirectoryFactory" class="solr.HdfsDirectoryFactory">
  <str name="solr.hdfs.home">hdfs://host:port/solr</str>
  <bool name="solr.hdfs.blockcache.enabled">true</bool>
  <int name="solr.hdfs.blockcache.slab.count">1</int>
  <bool name="solr.hdfs.blockcache.direct.memory.allocation">true</bool>
  <int name="solr.hdfs.blockcache.blocksperbank">16384</int>
  <bool name="solr.hdfs.blockcache.read.enabled">true</bool>
  <bool name="solr.hdfs.nrtcachingdirectory.enable">true</bool>
  <int name="solr.hdfs.nrtcachingdirectory.maxmergesizemb">16</int>
  <int name="solr.hdfs.nrtcachingdirectory.maxcachedmb">192</int>
</directoryFactory>
```

- ❑ solr.hdfs.blockcache.enabled：用于配置是否启用 HDFS Block 缓存，默认值为 true。
- ❑ solr.hdfs.blockcache.read.enabled：用于配置是否启用 HDFS Block 的读缓存，默认值为 true。
- ❑ solr.hdfs.blockcache.direct.memory.allocation：用于配置是否对 HDFS Block 启用直接内存分配，如果设置为 false，那么将直接使用 JVM 的堆内存进行分配，默认值为 true。
- ❑ solr.hdfs.blockcache.slab.count：用于配置分配的内存块个数，默认值 1。每个内存块大小默认为 128M。
- ❑ solr.hdfs.blockcache.global：启用或禁用 Solr 的所有 Core 的全局 Block 缓存。默认值为 true。
- ❑ solr.hdfs.nrtcachingdirectory.enable：用于配置是否启用 NRTCachingDirectory，默认值为 true。
- ❑ solr.hdfs.nrtcachingdirectory.maxmergesizemb：用于配置 NRTCachingDirectory 的最大段文件大小，默认值为 16，单位：MB。
- ❑ solr.hdfs.nrtcachingdirectory.maxcachedmb：用于配置 NRTCachingDirectory 的最大缓存大小，默认值为 192，单位：MB。
- ❑ solr.hdfs.confdir：用于配置 HDFS 客户端配置文件的加载路径，HDFS HA 会需要此配置参数。

为了提升性能，HdfsDirectoryFactory 会缓存 HDFS 中的 Block（数据块），HDFS 中一个 Block 默认大小为 128M。这种缓存机制意味着它会将 Solr 一直在使用的标准文件系统缓存替换为直接内存，直接内存的分配不受 Hava 内存大小限制，而是直接调用操作系统底层 API 来分配内存空间。但是可以通过传递下面这个 JVM 参数来限制直接内存空间的最大大小，以防止系统内存被无限撑爆：

```
-XX:MaxDirectMemorySize=20g
```


你需要明白什么时候应该使用 HDFS。我们都知道，HDFS 是为存储少量大文件而设计，而不是用于存储大量小文件，尤其是你的小文件需要频繁更新的场景下就更不适合了。这里说的小文件指的是文件大小小于 Hadoop 的默认 Block 大小（默认一个 Block 大小为 128M）的文件。如果你的索引数据频繁的在增量更新，那么存储在 HDFS 上的段文件就需要频繁的读写。大量的体积过小的段文件会极度消耗 namenode 的内存，而且为小文件单独执行一个 Map Task 是一种资源浪费。此时你需要确保你的 HDFS 执行开销不会过高。另一方面，如果你只是需要周期性的重建索引以及优化索引并存储到 HDFS 之上，然后索引更新不是那么频繁，在这种场景下，使用 HDFS 来存储 Solr 索引数据才能完美发挥 HDFS 的优势。

16.10 使用 Flume 收集数据并索引至 Solr

Flume 是一个基于流式的、健壮的、具有高容错性的数据收集框架，它简单易用，只需要写一个 source、channel、sink，然后执行一条命令就能完成数据收集操作。你可以将 Flume 收集到的数据写入到 Hive 或 Solr 等。下面请随我一起学习如何使用 Flume 收集日志数据并将其写入到 Solr 中。

假如有这样一段日志数据：

```
2016-11-14 20:09:05,915 [myid:1] - INFO [main:NIOServerCnxnFactory@94] -
binding to port 0.0.0.0/0.0.0.0:2181
```

2016-11-14 20:09:05,915 部分表示日志打印时间，[myid:1] 部分表示 Zookeeper 的 service id 值，INFO 表示日志输出级别，后面的 [main:NIOServerCnxnFactory@94] 表示当前日志信息由哪个类打印的，最后一部分就是实际的日志信息。想要这些日志信息写入 Solr 中，我们首先需要创建一个 Collection（如果 Solr 单机模式下，就创建一个 Core），这里暂且命名为 logs，并且为 logs 创建 schema.xml 和 solrconfig.xml，其中 schema.xml 部分定义如下所示：

```
<fieldType name="uuid" class="solr.UUIDField" indexed="true"/>
<fields>
  <field name="id" type="uuid" indexed="true" stored="true" required="true"
multiValued="false"/>
  <field name="zk_server_id" type="string" indexed="true" stored="true"/>
  <field name="log_datetime" type="string" indexed="true" stored="true"/>
  <field name="log_level" type="string" indexed="true" stored="true"/>
  <field name="log_class" type="string" indexed="true" stored="true"/>
  <field name="log_msg" type="string" indexed="true" stored="true"/>
  <field name="_version_" type="long" indexed="true" stored="true" /><uniqueKey>id</
uniqueKey>
```

然后执行如下命令创建我们的“logs”Collection：

```
solr create_collection -c logs -d /opt/solr_home/logs/conf -shards 3
```

```
-replicationFactor 4
```

开始安装 Flume。下面是 Flume 安装包的下载地址：

#Apache Flume 下载地址：

<http://archive.apache.org/dist/flume/>

#Cloudera CDH Flume 下载地址：

<http://archive.cloudera.com/cdh5/>

这里我将以 Apache 版的 Flume1.7.0 为例进行讲解。下面是 Flume 的详细安装过程：

```
sudo tar -zxf apache-flume-1.7.0.tar.gz -C /opt/modules/
cd /opt/modules/
sudo mv apache-flume-1.7.0-bin/ apache-flume-1.7.0
sudo chown -R hadoop:hadoop apache-flume-1.7.0
```

为了便于在任意路径下都能执行 Flume 命令，这里我建议你配置 Flume 环境变量：

```
sudo vi /etc/profile
#FLUME_HOME
FLUME_HOME=/opt/modules/apache-flume-1.7.0
在 PATH 环境变量末尾追加 :$FLUME_HOME/bin
source /etc/profile
```

紧接着你需要配置 Flume 的 conf 目录下的 flume-env.sh 文件，主要是在其中配置一下 JAVA_HOME，为了防止在执行 Flume 命令过程中出现 OutOfMemory 异常，请同时为 JVM 配置一下堆内存，其他配置都是可选的：

```
cd /opt/modules/apache-flume-1.7.0/conf
mv flume-env.sh.template flume-env.sh
sudo vi flume-env.sh
export JAVA_HOME=/opt/modules/jdk1.8.0_111
export JAVA_OPTS="-Xms512m -Xmx1024m"
```

最后，请使用 flume-ng version 命令验证 Flume 是否安装成功。到此，Flume 就安装成功了。紧接着我们需要对 Flume 进行配置将其与 Solr 进行集成。

首先请在 Flume 的 conf 配置目录下通过 touch 命令新建一个 flume-solr.properties 配置文件，配置示例如下所示：

```
a1.channels = logsChannel
a1.sources = logs
a1.sinks = logsSink
a1.channels.logsChannel.type = memory
a1.sources.logs.channels = logsChannel
a1.sources.logs.type = exec
a1.sources.logs.command = tail -F /var/log/zk.log
a1.sinks.logsSink.morphlineId = morphline1
a1.sinks.logsSink.channel = logsChannel
a1.sinks.logsSink.type = org.apache.flume.sink.solr.morphline.MorphlineSolrSink
```

```

a1.sinks.logsSink.morphlineFile = /opt/modules/apache-flume-1.7.0/conf/
morphline.conf
a1.sinks.logsSink.batchSize = 1
a1.sinks.logsSink.batchDurationMillis = 1000

```

上面配置中有 3 个关键点，第一点就是 `a1.sources.logs.command` 配置，它使用 linux 的 `tail-F` 命令增量的读取日志文件，因此此时你应该在 `/var/log/` 目录下创建一个 `zk.log` 测试文件并插入一条测试数据。第二点就是 `a1.sinks.logsSink.type` 配置，它指定使用 `MorphlineSolrSink` 将 Flume 收集到的数据写入到 Solr 中。最后一点就是 `a1.sinks.logsSink.morphlineFile` 配置，它用于配置 `morphline` 依赖的配置文件。`morphline.conf` 的配置示例如下所示：

```

SOLR_LOCATOR: {
  collection : logs
  zkHost : "linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181"
}
morphlines : [
  {
    id : morphline1
    importCommands : ["org.kitesdk.**", "org.apache.solr.**"]
    commands : [
      {
        readLine {
          charset : UTF-8
        }
      }
      {
        grok {
          dictionaryFiles : [/opt/modules/apache-flume-1.7.0/conf/grok-dictionaries]
          expressions : {
            message : """"${TIMESTAMP_LOG:log_datetime} \[myid:${ZK_SERVER_ID:zk_server_id}\] \- ${LOGLEVEL:log_level} \[main:${DATA:log_class}\] \- ${GREEDYDATA:log_msg}""""
          }
        }
      }
      {
        convertTimestamp {
          field : log_datetime
          inputFormats : ["yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", "yyyy-MM-ddHH:mm:ss,SSS"]
          inputTimezone : America/Los_Angeles
          outputFormat : "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"
          outputTimezone : UTC
        }
      }
      {
        generateUUID {
          field : id
        }
      }
    ]
  }
]

```

```

    }
    {
        sanitizeUnknownSolrFields {
            solrLocator : ${SOLR_LOCATOR}
        }
    }
    { logInfo { format : "output record: {}", args : ["@{}"] } }
    {
        loadSolr {
            solrLocator : ${SOLR_LOCATOR}
        }
    }
}
}
1

```

上述配置中 `dictionaryFiles` 用于配置 Flume 正则表达式文件所在的目录，关于这部分配置文件请从随书源码中获取。`expressions` 部分用于定义如何解析日志文件中的每一行数据，主要采用 `%{TIMESTAMP_LOG:log_datetime}` 来解析数据，`TIMESTAMP_LOG` 是提前在 `dictionaryFiles` 目录下的正则表达式文件中定义的正则表达式对应的引用名称，用户可以随意在其中自定义正则表达式。`log_datetime` 表示解析出来的数据写入到 Solr 的哪个域中即我们在 `schema.xml` 中定义的域名称。此外特别提醒，`message` 参数值中如果需要包含正则表达式的特殊字符，比如上面示例中 `[` 和 `-` 字符，都属于正则表达式中的特殊字符，因此都使用了反斜杠 `\` 进行了转义。`convertTimestamp` 部分配置用于定义日志文件中的日期格式，便于 Flume 进行日期数据解析。`generateUUID` 用于配置 Flume 自动生成 UUID 作为 Solr 的主键 `id` 域的域值。`sanitizeUnknownSolrFields` 用于配置域名称检查，即 Flume 需要用户配置的 Zookeeper 访问地址，从 Zookeeper 上获取 `schema.xml` 文件，从而实现对用户输入的 Solr 域名称进行校验。`loadSolr` 部分配置主要用于 Flume 与 Solr 进行交互需要的参数，因此 Flume 需要将解析出来的数据写入到 Solr，因此 Flume 需要知道 Zookeeper 集群的访问地址以及 Collection 名称。

我们需要启动 Solr 集群，再启动 Flume。请执行如下命令来启动 Flume：

```
flume-ng agent -c conf -f conf/flume-solr.properties -n a1 -Dflume.root.logger=INFO,console
```

不过在启动之前，你需要从 Solr 安装包中查找 Flume 与 Solr 集成所依赖的 jar 包添加到 Flume 的 `lib` 目录下，依赖的 jar 包文件如下所示：

```

solr-6.2.1\contrib\morphlines-cell\lib 目录下：
kite-morphlines-json-1.1.0.jar
kite-morphlines-twitter-1.1.0.jar
solr-6.2.1\contrib\morphlines-core\lib 目录下：
kite-morphlines-core-1.1.0.jar
kite-morphlines-avro-1.1.0.jar

```

```

metrics-core-3.0.1.jar
metrics-healthchecks-3.0.1.jar
config-1.0.2.jar
solr-6.2.1\dist 目录下:
solr-core-6.2.1.jar
solr-solrj-6.2.1.jar
solr-6.2.1\dist\solrj-lib 目录下:
zookeeper-3.4.6.jar
noggit-0.6.jar
solr-6.2.1\server\solr-webapp\webapp\WEB-INF\lib 目录下:
lucene-core-6.2.1.jar
lucene-analyzers-common-6.2.1.jar
lucene-queries-6.2.1.jar
commons-fileupload-1.3.1.jar
httpclient-4.4.1.jar
httpcore-4.4.1.jar
httpmime-4.4.1.jar

```

同时,记得删除 Flume 的 lib 目录下原本已经存在的两个 jar 包: httpclient-4.2.1.jar 和 httpcore-4.1.3.jar。因为 Solr6 依赖的是 httpclient-4.4.1.jar 和 httpcore-4.4.1.jar。如果需要使用 Flume 来提取 txt、word、pdf 等文件并导入到 Solr 创建索引,此时你还需要额外添加 solr-6.2.1\contrib\morphlines-cell\lib 目录下的两个 jar 包:

```

kite-morphlines-tika-core-1.1.0.jar
kite-morphlines-tika-decompress-1.1.0.jar

```

启动 Flume 之后,Flume 就会实时监控 /var/log 目录下的 zk.log 文件,并根据我们定义的解析表达式提取日志数据,写入到 Solr 中。然后你打开 Solr 的后台管理界面就能查询到索引数据。至此,Flume 与 Solr 集成就介绍到这里了。关于本章节示例的完整代码与配置文件,请从随书源码的第 16 章中获取。当然 Flume 不仅仅可以与 Solr 集成,你还可以将其与 MySQL 集成,Flume 实时的解析 MySQL 的事务日志来获取数据库表中的增量数据,然后导入到 Solr 中,从而实现 Solr 索引数据的自动增量更新,关于这个话题限于篇幅不便继续展开,就留给大家自己去探索学习了。

16.11 使用 Solr 实现 HBase 的二级索引

HBase 即 Hadoop Database,是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统,利用 HBase 技术可在廉价 PC Server 上搭建起大规模结构化存储集群。HBase 是建立在 HDFS 之上的列式存储数据库,HDFS 保证了 HBase 适合存储 PT 级数据。HBase 的存储结构松散,列可以动态添加,决定了 HBase 更能满足复杂多变的业务需求。HBase 本身的数据读写服务没有单点限制,服务能力可以随服务器的增加而线性增长,可以达到几十上百台的规模。LSM-Tree 模式设计让 HBase 的写入性能非常良好,单次写入通常在 1-3ms 内即

可响应完成，且性能不随数据量的增长而下降。HBase 的 Region（相当于数据库的分表）可以毫秒级动态切分和移动，保证了负载均衡性。由于 HBase 上的数据模型是按 rowkey 排序存储的，读取时会一次读取连续的整块数据作为 cache，因此良好的 rowkey 设计可以让批量读取变得十分容易，甚至只需要 1 次 IO 就能获取几十上百条用户想要的数据库。

由于 Hbase 只有根据 Rowkey 查询性能才高效，根据其他 Column 查询就会导致全表扫描，因此一般需要根据其他列为 Hbase Table 创建二级索引，就好比我们为 MySQL 的 Table 创建索引是类似的道理，主要就是为了实现类似 SQL 里的多条件组合查询性能高效。常规的做法是为 HBase 创建一个二级索引表，查询之前先扫描二级索引表得到 rowkey，再根据 rowkey 去主表获取该条记录。关于这种方式，华为提供了一个开源实现：<https://github.com/Huawei-Hadoop/hindex/>。不过遗憾的是，已经 3 年多没更新了，目前支持的 HBase 版本为 Apache HBase 0.94.8。如果你想要升级到当前 HBase 最新版本，由于 HBase API 变化有点大，这个升级工作难度有点大。不过这里我推荐你使用 NGDATA 开源的 Lily HBase Indexer 项目，它支持使用 Solr 作为 HBase 的二级索引。

HBase Indexer 项目支持将存储在 HBase 上的数据索引到 Solr 中，Solr 索引过程是异步执行的，因此它不会影响 HBase 写操作的吞吐量。SolrCloud 用于存储实际的索引，用于确保索引的可伸缩性。HBase Indexer 通过 HBase 的复制功能来实现，当数据写入 HBase Region 时，数据会被异步复制到 HBase 索引协处理器，索引优化器创建文档并推送到 SolrCloud Server 中。索引在 Solr 中的 Document 拥有足够的信息来唯一标示存储在 HBase 的 Row，这样你就能够使用 Solr 查询存储在 HBase 中的内容。关于该开源项目的一些相关链接如下所示：

```
http://ngdata.github.io/hbase-indexer/ // 项目官网
https://github.com/NGDATA/hbase-indexer // Lily HBase Indexer 项目在 Github 上的托管地址
https://github.com/NGDATA/hbase-indexer/wiki/Tutorial // Lily HBase Indexer 项目的 Wiki 页面
```

在我编写本章时，Lily HBase Indexer 已经支持 HBase1.1，Solr6.2.0 版本，所以可以放心使用，不必担心因为版本过低而无法兼容的问题，而且源码编译简单，只需要从 Github 上下载其源码，然后执行如下命令进行编译：

```
mvn install -e -Dmaven.test.skip=true-Dhbase.api=1.2.4
```

-Dhbase.api=1.2.4 部分表示你需要基于哪个 HBase 版本进行编译。不过，为了使得编译过程更顺畅，这里我建议你 Maven 的 settings.xml 中配置国内的阿里云仓库，配置示例如下所示：

```
<mirror>
<id>nexus-aliyun</id>
<mirrorOf>*</mirrorOf>
```



```
<name>Nexus aliyun</name>
<url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

编译成功之后，请在项目根目录下的 pom.xml 中添加 maven-dependency-plugin 插件，使用该插件来提取项目依赖的所有 jar 包，配置示例如下所示：

```
<plugin>
  <artifactId>maven-dependency-plugin</artifactId>
  <configuration>
    <outputDirectory>${project.build.directory}/lib</outputDirectory>
    <excludeTransitive>false</excludeTransitive>
    <stripVersion>true</stripVersion>
  </configuration>
</plugin>
```

然后执行 mvn dependency:copy-dependencies 命令进行 jar 包提取，这个过程可能会持续 5 ~ 10 分钟左右，请耐心等待。

请在 D 盘（其他盘符也行）新建一个 hbase-indexer-1.6 目录，将源码根目录下的 bin、conf、hbase-indexer-all 这 3 个目录全部复制到我们刚刚新建的 hbase-indexer-1.6 目录下，同时在 hbase-indexer-1.6 目录下新建 lib 目录，在源码根目录下 Ctrl + f 搜索 *.jar 找出所有 jar 包，Ctrl + a 复制到我们刚刚新建的 lib 目录下。在 hbase-indexer-1.6 目录上鼠标右键→添加到压缩文件，这里必须压缩为 zip 格式，因为 rar 格式在 linux 上默认不支持解压。你将得到一个 hbase-indexer-1.6.zip 压缩文件，最后将其上传至 Linux 服务器。你需要先安装 HBase，随后再安装 hbase-indexer。关于 HBase 如何安装这里就不介绍了，网上有很多这方面的安装教程，请自己上网查阅资料完成 HBase 的安装。由于 HBase 依赖 Zookeeper，因此在安装 HBase 之前你需要先安装 Zookeeper 集群。在安装完 HBase 之后，请随即安装部署 SolrCloud 环境，因为 HBase Indexer 只支持 SolrCloud，不支持 Solr 单机版。

这里我假定你已经完成了 HBase 的安装。下面请你跟随我一起动手完成 HBase Indexer 的安装部署。

首先请解压 HBase Indexer 安装包，并赋予它相关权限，执行命令如下所示：

```
sudo unzip hbase-indexer-1.6.zip -d /opt/modules/
cd /opt/modules/
sudo chown -R hadoop:hadoop hbase-indexer-1.6
```

配置 HBase Indexer 的环境变量，配置示例如下所示：

```
HBASE_INDEXER_HOME=/opt/modules/hbase-indexer-1.6
```

请记得在 PATH 环境变量末尾追加 :\$ HBASE_INDEXER_HOME/bin，最后别忘了执行 source/etc/profile 使上述配置立即生效。配置 hbase-indexer-1.6/conf 目录下的 hbase-indexer-env.sh：

```
export JAVA_HOME=/opt/modules/jdk1.8.0_111
export HBASE_INDEXER_HEAPSIZE=1024
export HBASE_INDEXER_LOG_DIR=${HBASE_INDEXER_HOME}/logs
export HBASE_INDEXER_PID_DIR=/opt/modules/hbase-indexer-1.6/pid
export HBASE_INDEXER_CLI_ZK=linux.yida01.com,linux.yida02.com,linux.yida03.com
```

在 /opt/modules/hbase-indexer-1.6/ 目录下新建 logs 和 pid 目录。配置 hbase-indexer-1.6/conf 目录下的 hbase-indexer-site.xml:

```
<configuration>
<property>
<name>hbaseindexer.defaults.for.version.skip</name>
<value>true</value>
</property>
  <property>
    <name>hbaseindexer.zookeeper.connectstring</name>
    <value>linux.yida01.com,linux.yida02.com,linux.yida03.com</value>
  </property>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>linux.yida01.com,linux.yida02.com,linux.yida03.com</value>
  </property>
</configuration>
```

配置 HBase 的 conf 目录下的 hbase-site.xml, 为其开启复制机制:

```
<property>
<name>hbase.replication</name>
<value>true</value>
</property>
<property>
  <name>replication.source.ratio</name>
  <value>1.0</value>
</property>
<property>
  <name>replication.source.nb.capacity</name>
  <value>1000</value>
</property>
<property>
  <name>replication.replicationsource.implementation</name>
  <value>com.ngdata.sep.impl.SepReplicationSource</value>
</property>
```

将 HBase Indexer 的一些 jar 包复制到 \${HBASE_HOME}/lib 目录下, 这里假定你的 \$HBASE_HOME 为 /opt/modules/hbase-1.2.4, 执行命令如下所示:

```
/opt/modules/hbase-indexer-1.6/lib
cp *.jar /opt/modules/hbase-1.2.4/lib/
```

然后执行 hbase-indexer server 命令启动我们的 HBase Indexer 服务, 执行完成之后, 请

另起一个命令行窗口执行 `jps` 命令查看是否有一个名称为 `Main` 的进程存在，如果存在则表明 `HBase Indexer` 启动成功了。不过启动之前确保你的 `Zookeeper` 集群已经启动。

使用 `hbase shell` 命令进入 `HBase Shell` 命令行，这里我们创建一个名称为 “`user-test`” 的测试表：

```
create 'user-test', { NAME => 'info', REPLICATION_SCOPE => '1' }
```

其中 `user-test` 为我们定义的 `HBase` 表名称，`info` 为列簇名称，这里 `REPLICATION_SCOPE` 属性必须设置为 1。

现在我们需要创建一个 `Indexer`，当 `user-test` 表的数据更新了，`Indexer` 就会对 `user-test` 表创建 `Solr` 索引。请使用 `touch` 命令创建一个 `xml` 文件，这里暂且命名为 `user-test-indexer.xml`，`xml` 内容编辑如下：

```
<?xml version="1.0"?>
<indexer table="user-test" unique-key-field="id">
<field name="firstname" value="info:firstname"/>
<field name="lastname" value="info:lastname"/>
<field name="age" value="info:age" type="int"/>
</indexer>
```

上面的 `table` 是我们创建的表名称，`unique-key-field` 属性对应 `Solr` 的 `schema.xml` 中配置的 `UniqueKey` 域。`<field>` 是用来定义 `Solr` 里的域，`name` 属性表示 `Solr` 的 `schema.xml` 中已经定义的域名称，`value` 表示当前域的域值来自 `HBase` 表的哪个列簇的哪个列，`info` 为我们定义的列簇名称，`firstname`、`lastname`、`age` 分别为 `info` 这个列簇下的 3 个列。即我们定义了需要将 `HBase` 的 `user-test` 表的 `info` 列簇下的 `firstname`、`lastname`、`age` 这 3 列数据索引到 `Solr` 中。`type` 属性用于定义域类型。因此，此时我们应该立马去创建一个 `Collection`，定义好 `firstname`、`lastname`、`age` 这 3 个域。首先你需要启动 `SolrCloud` 集群，然后执行如下命令创建一个名称为 “`user-test`” 的 `Collection`：

```
solr create_collection -c user-test -d /opt/solr_home/user-test/conf -shards 3
-replicationFactor 4
```

紧接着请创建 `schema.xml` 并定义上述的 3 个域，配置示例如下所示：

```
<fields>
<field name="id" type="uuid" indexed="true" stored="true" required="true"
multiValued="false"/>
<field name="firstname" type="string" indexed="true" stored="true"/>
<field name="lastname" type="string" indexed="true" stored="true"/>
<field name="age" type="int" indexed="true" stored="true"/>
<field name="_version_" type="long" indexed="true" stored="true" />
</fields>
<uniqueKey>id</uniqueKey>
```

下面我们开始根据上面创建的 `user-test-indexer.xml` 文件来创建我们的 `Indexer`，执行命

令如下所示：

```
hbase-indexer add-indexer -n myindexer -c
/opt/modules/hbase-indexer-1.6/conf/user-test-indexer.xml -cp
solr.zk=linux.yida01.com,linux.yida02.com,linux.yida03.com -cp
solr.collection=user-test -cp solr.mode=cloud
```

其中 `-n` 表示创建的索引在 Zookeeper 上的名称，稍候我们可以根据此名称来删除我们创建的索引，`-c` 参数用于配置我们创建 Indexer 依赖的 XML 文件。`solr.zk` 用于配置 HBase Indexer 使用的 Zookeeper 集群，`solr.collection` 参数表示创建的索引存储到 Solr 的哪个 Collection 中，此 Collection 必须提前存在。`solr.mode` 表示当前是 SolrCloud 模式，而非 Solr 单机模式。

HBase Indexer 除了 `add-indexer` 命令之外，还支持如下几个命令：

```
$ hbase-indexer add-indexer --help
$ hbase-indexer list-indexers --help
$ hbase-indexer update-indexer --help
$ hbase-indexer delete-indexer --help
```

我们的索引已经创建成功了，现在我们尝试在 HBase Shell 命令行下插入一条测试数据到 HBase 表中，执行命令如下所示：

```
put 'user-test', '001', 'info:firstname', 'John'
put 'user-test', '001', 'info:lastname', 'Smith'
```

上面的命令中，我们往 HBase 的 “user-test” 表的 `info` 这个列簇的 `firstname` 和 `lastname` 两列分别插入了数据，同时指定 `rowkey=001`，执行之后，你可以通过 `get 'user-test','001'` 命令检查我们刚刚插入的数据是否插入成功。如果 Hbase 中数据插入成功了，那么你将会在 Solr 的后台管理界面中查询到我们刚刚在 HBase 中插入的数据。不过这里有个前提就是需要在 Solr 的 `solrconfig.xml` 中配置自动软提交，否则你将看不到数据，除非重新加载 Core 或 Collection 才能看到 HBase 最新插入的数据。此时 Solr 中的数据就作为 HBase 的二级索引，如果需要对 HBase 中的数据根据其他列进行查询，直接查询 Solr 即可。可能 HBase 中的部分列没有存储到 Solr 索引中，可是我们可以根据已经存储到 Solr 的列进行查询得到 Rowkey，然后你拿到 Rowkey 再在 HBase 表中查询就很快了。这样你就可以基于 HBase 表中任意列进行条件组合查询，而不仅仅限制于 Rowkey 查询。

16.12 Solr 与 Kafka、Flume 集成

Apache Kafka 是一款由 Linked-in 开源的基于发布 - 订阅模式的具有高吞吐量的分布式消息系统，它具备快速、可扩展、可持久化的特点。它目前是 Apache 旗下的一个开源系统，作为 Hadoop 生态系统的一部分，被各大商业公司广泛应用。它的最大的特性就是可以

实时的处理大量数据以满足各种需求场景：比如基于 hadoop 的批处理系统、低延迟的实时系统、Storm/Spark 的流式处理引擎。

Kafka 的主要使用场景如下：

- ❑ 日志数据收集：可以使用 Kafka 收集各种系统服务的日志数据，然后通过 Kafka 以统一接口服务的方式开放给各种 Consumer，比如 Hadoop、HBase、Solr。
- ❑ 消息系统：可以作为类似 ActiveMQ 的分布式消息系统，不过与 ActiveMQ 不同的是，Kafka 并不遵循 JMS 规范。
- ❑ 用户行为追踪：可以使用 Kafka 来记录 Web 用户或手机 APP 用户的各种行为活动，比如浏览网页痕迹、搜索行为、单击行为等。这些用户行为数据会被各个服务器发布到 Kafka 的 Topic 中，然后订阅者可以订阅此 Topic 来实时监控以及数据分析，比如使用 Hadoop、Hive、Spark 进行数据分析。
- ❑ 运营指标：经过上一步中对用户行为数据的分析，可以生产各种反馈数据，比如 PV、UV。
- ❑ 流式处理：Spark Streaming、Storm。

这里我们主要讲解的就是第一种使用场景，即使用 Kafka 来收集日志数据并创建 Topic 将日志数据发布到消息系统，然后 Flume 订阅该 Topic 消费该日志数据，最后 Flume 将日志数据索引至 Solr 中，下面请跟随我一起动手实践，完成 Solr 与 Kafka、Flume 的集成。不过在开始之前，请你首先安装好 SolrCloud 集群和 Flume。

假定此时你已经完成 SolrCloud 集群和 Flume 的安装，下面紧接着我们需要安装 Kafka，这里以当前最新版本 kafka_2.11-0.10.1.0 为例进行讲解。Kafka 下载地址如下：

<https://kafka.apache.org/downloads>

Kafka 的详细安装过程如下所示：

```
# 解压 Kafka 安装包
sudo tar -zxvf kafka_2.11-0.10.1.0.tgz -C /opt/modules/
# 赋予 Kafka 的用户所有者为 hadoop 用户，一般不会直接使用 root 用户直接操作
cd /opt/modules/
sudo chown -R hadoop:hadoop kafka_2.11-0.10.1.0/
```

跟其他框架类似，这里我建议你为 Kafka 配置环境变量，方便在任意路径下都能执行 Kafka 的脚本命令：

```
KAFKA_HOME=/opt/modules/kafka_2.11-0.10.1.0
```

配置 config 目录下的 server.properties 属性文件：

```
broker.id=0
listeners=PLAINTEXT://linux.yida01.com:9092
log.dirs=/opt/modules/kafka_2.11-0.10.1.0/logs
num.partitions=2
```

```
zookeeper.connect=linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181
```

其中 `broker.id` 用于配置 `broker` 的唯一 `id`，必须保证全局唯一。`Listeners` 用于配置当前 `Kafka` 服务监听的主机 `ip` (或域名) 与端口号。`log.dirs` 用于配置 `Kafka` 的日志文件存放目录，`num.partitions` 用于配置 `Kafka` 日志文件的分区数，默认值为 1。`zookeeper.connect` 用于配置 `Kafka` 服务使用的 `Zookeeper` 访问字符串，多个 `Zookeeper` 节点的配置采用逗号分隔，默认端口号 2181 可以省略不配置。执行如下命令启动 `Kafka` 服务：

```
kafka-server-start.sh config/server.properties &
```

执行 `jps` 命令检查 `Kafka` 服务是否正常启动，如果能够看到一个名称为 `Kafka` 的系统进程，则表明 `Kafka` 服务已经启动成功。

编写一个 `Flume` 配置文件，示例如下：

```
flume1.sources = kafka-source-1
flume1.channels = mem-channel-1
flume1.sinks = solr-sink-1
flume1.sources.kafka-source-1.type = org.apache.flume.source.kafka.KafkaSource
flume1.sources.kafka-source-1.zookeeperConnect = linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181
flume1.sources.kafka-source-1.kafka.bootstrap.servers = linux.yida01.com:9092
flume1.sources.kafka-source-1.kafka.topics = test-topic
flume1.sources.kafka-source-1.batchSize = 100
flume1.sources.kafka-source-1.channels = mem-channel-1
flume1.channels.mem-channel-1.type = memory
flume1.sinks.solr-sink-1.channel = mem-channel-1
flume1.sinks.solr-sink-1.type = org.apache.flume.sink.solr.morphline.MorphlineSolrSink
flume1.sinks.solr-sink-1.batchSize = 100
flume1.sinks.solr-sink-1.batchDurationMillis = 1000
flume1.sinks.solr-sink-1.morphlineFile = /opt/modules/apache-flume-1.7.0/conf/kafka-morphline.conf
flume1.sinks.solr-sink-1.morphlineId = morphline2
```

上述配置中有几个关键点需要稍作说明，如下所示：

`sources` 的 `type` 定义为 `KafkaSource`，表示数据源从 `Kafka` 获取，`sources` 的 `zookeeperConnect` 参数用于配置 `Kafka` 使用的 `Zookeeper` 集群访问字符串，`kafka.bootstrap.servers` 用于配置 `Kafka` 服务所在主机 `ip` (或域名) 以及监听的端口号。`kafka.topics` 用于配置 `Flume` 订阅 `Kafka` 的哪些 `Topic`，多个 `Topic` 使用逗号分隔。`sinks` 的 `type` 定义为 `MorphlineSolrSink` 表示 `Flume` 会把从 `Kafka` 的 `Topic` 订阅获取到的数据索引至 `Solr`，`sinks` 的 `morphlineFile` 配置是 `Flume` 收集到的数据能够索引至 `Solr` 的关键，我们需要在 `kafka-morphline.conf` 配置文件中定义数据的解析规则。关于 `kafka-morphline.conf` 配置文件的编写，我们已经在 16.10 节中接触过，这里不再详述。这里可以复制之前的配置，然后稍作修改即可，配置示例如下：


```

SOLR_LOCATOR: {
  collection : user-test
  zkHost : "linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.
com:2181"
}
morphlines : [
  {
    id : morphline2
    importCommands : ["org.kitesdk.**","org.apache.solr.**"]
    commands : [
      {
        readLine {
          charset : UTF-8
        }
      }
      {
        grok {
          dictionaryFiles : [/opt/modules/apache-flume-1.7.0/conf/grok-
dictionaries]
          expressions : {
            message : ""${DATA:id} ${DATA:
firstname} ${DATA:lastname} ${DATA:age}""
          }
        }
      }
      {
        sanitizeUnknownSolrFields {
          solrLocator : ${SOLR_LOCATOR}
        }
      }
      { logInfo { format : "output record: {}", args : ["@{}"] } }
      {
        loadSolr {
          solrLocator : ${SOLR_LOCATOR}
        }
      }
    ]
  }
]

```

我们在前面的 Flume 配置文件中配置了 sources 订阅 Kafka 的名称为 test-topic 的 Topic，因此现在我们需要执行 Kafka 提供的脚本文件创建该 Topic，执行命令如下所示：

```
kafka-topics.sh --create --topic test-topic --partitions 1 --replication-factor
1 --zookeeper linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181
```

启动 Flume 来订阅 Kafka 的 Topic 收集数据，执行命令如下所示：

```
cd /opt/modules/apache-flume-1.7.0/
flume-ng agent -c conf -f conf/flume-kafka.properties -n flume1 -Dflume.root.
logger=INFO,console
```

使用 Kafka 提供的 kafka-console-producer.sh 脚本文件开启一个 Kafka 消费者，通过命令行的方式向“test-topic”这个 Topic 生产消息，由于 Flume 订阅了该 Topic，因此 Flume 能够实时收集到该消息，同时将收集到的消息数据索引至 Solr。随后你就能够在 Solr 中查

询到该消息数据。当然这里仅仅只是通过命令行的方式模拟消息生产者，实际项目你可能需要通过 Kafka 的 API 自己开发 Kafka 消息生产者或消费者。至此，Solr 与 Kafka、Flume 集成实现分布式实时系统的雏形就介绍到这里了。关于 Kafka、Flume 更详细的内容建议读者阅读《Kafka Cookbook》与《Apache Flume Distributed Log Collection for Hadoop》这两本书。

16.13 使用 Storm 索引数据至 Solr

如今我们已经步入科技高速发展的信息时代，信息数据呈现爆发式增长，从而催生了大数据领域的高速发展。说到大数据，不得不提大数据生态系统中大名鼎鼎的 Hadoop、Hive、HBase。Hadoop 以其高吞吐量、自动容错性、分布式离线计算而著称。但是 Hadoop 的数据处理是离线延迟的，不符合实时性要求高的场景。因此，Puma、Storm、Spark 这类实时分布式计算框架又如雨后春笋般不断涌现出来，并且大受追捧。

Apache Storm 是一个开源免费的实时分布式计算系统框架，Storm 使源源不断的实时可靠的处理流式数据变得简单。Storm 自身非常简单，它可以与任何编程语言搭配使用，并且其乐无穷。

Storm 拥有很多使用案例：实时数据分析、在线机器学习、连续计算、分布式 RPC、ETL 等。Storm 运行非常快速：Storm 的标准记录是每台节点每秒能够处理超过 100 万的 Tuple。同时，Storm 还具有可伸缩性、自动容错性，保证你的数据一定会被处理，并且，Storm 安装与操作非常简单。

Storm 集成了我们使用过的队列与数据库技术。一个 Storm topology 消费流式数据，然后使用任意复杂的方式处理这些流式数据，同时按需在计算的各个 Stage（阶段）之间重新分配这些流式数据。

下面我将带领大家以实际案例的方式完成将 Storm topology 的对象发送至 Solr 进行索引的流程。

首先安装好 SolrCloud 环境，然后安装 Storm 集群。这里我们以当前 Storm 最新版本 1.0.2 为例进行讲解说明。请从 Storm 官网下载获取 Storm 的安装包，下载地址如下：

<http://storm.apache.org/downloads.html>

搭建一个 Storm 集群大致需要以下几个步骤：

- ☐ 搭建 Zookeeper 集群。
- ☐ 请在 worker 和 Nimbus 机器上安装 Storm 相关依赖，比如 JDK1.6+、Python2.6+。
- ☐ 下载并解压 Storm 发布版安装包到 worker 和 Nimbus 机器上。
- ☐ 修改 storm.yaml 配置文件。
- ☐ 使用 "storm" 脚本启动运行 Storm 的各个后台进程。

这里跳过第一步搭建 Zookeeper 集群的步骤，然后通过 java-version 命令检查当前机器

安装的 JDK 版本, 如果 JDK 版本低于 1.6, 那么请安装更高版本的 JDK。然后通过 python-V 命令检查当前机器上安装的 Python 版本, 如果 Python 版本低于 2.6, 请从 Python 官网下载获取更高版本的 Python 安装包, Python 安装包下载地址: <https://www.python.org/ftp/python/>。这里以 Python2.7.0 为例。首先请使用 rpm-e--nodeps python 命令卸载系统自带的旧版本 Python。然后从 Python 官网下载安装包, Python 官网下载地址为: <https://www.python.org/ftp/python/>。解压 Python 安装包:

```
gunzip Python-2.7.tgz&&tar -xf Python-2.7.tar -C /opt/modules/
```

进入 Python-2.7 目录, 对 Python 进行编译安装, 具体执行命令如下所示 (注意请使用 root 账号执行如下命令):

```
mkdir -p /opt/modules/python2.7.0      // 此目录作为 Python 的自定义安装目录
yum -y install make gcc-c++ cmake      // 确保你已经安装了 gcc 编译器
./configure --prefix=/opt/modules/python2.7.0&&make && make install
```

配置 Python 的环境变量:

```
PYTHON_HOME=/opt/modules/python2.7.0
```

同时在 PATH 环境变量末尾追加 :\$PYTHON_HOME/bin, 执行 source/etc/profile 命令使其立即生效, 最后执行 python-V 检查 Python 当前版本是否为 2.7。然后你需要在 Storm 集群中的其他节点上也安装 Python2.7.0 环境。

将 Storm 安装包上传到 Master 机器, 这里假定你的 Master 机器的域名为 linux.yida01.com。Storm 的详细安装过程如下:

```
sudo tar -zxf apache-storm-1.0.2.tar.gz -C /opt/modules/ // 解压安装包至 /opt/modules/
                                                         // 目录下
cd /opt/modules/apache-storm-1.0.2/conf
vi storm.yaml                                           // 编辑 storm.yaml 配置文件
storm.zookeeper.servers:                               // 配置 Zookeeper 集群
- "linux.yida01.com"
- "linux.yida02.com"
- "linux.yida03.com"
nimbus.seeds: ["linux.yida01.com"]                     // 配置 Storm Master 节点的 ip 或域名
supervisor.slots.ports:
- 6700
- 6701
- 6702
- 6703
storm.local.dir: "/opt/modules/apache-storm-1.0.2/data/" // 配置 Storm 的 Nimbus 和 Supervisor 进程数据存储目录, 需要提前创建并赋予足够的操作权限, 请记得在集群的每个节点都创建此目录。
```

配置 conf 目录下的 storm-env.sh 配置文件, 配置示例如下:

```
export JAVA_HOME=/opt/modules/jdk1.8.0_111
```

对 bin 目录下的 storm 脚本文件稍作编辑, 在其中配置 Python 的安装路径, 配置示例

如下所示:

```
PYTHON=/opt/modules/python2.7.0/bin/python
# find python >= 2.6
if [ -a /usr/bin/python2.6 ]; then
    PYTHON=/usr/bin/python2.6
fi
```

执行 scp 命令将 Master 节点上安装好的 Storm 分发到 Storm 集群中的其他节点上, 执行之前请确保你已经配置了集群各节点之间的免密码登录, 执行命令如下所示:

```
sudo scp -r /opt/modules/apache-storm-1.0.2 linux.yida02.com:/opt/modules/
sudo scp -r /opt/modules/apache-storm-1.0.2 linux.yida03.com:/opt/modules/
```

为了使我们在执行 Storm 的脚本命令时不需要每次都切换到 Storm 的 bin 目录下, 这里我建议你将 Storm 配置系统环境变量, 配置示例如下:

```
STORM_HOME=/opt/modules/apache-storm-1.0.2
```

请记得在 PATH 环境变量末尾追加 :\$STORM_HOME/bin, 最后你需要执行 source/etc/profile 使其立即生效, 同时不要忘记在 Storm 集群的其他节点上也配置 STORM_HOME。

在 Master 节点上启动 Storm 的 Nimbus 进程, 执行命令如下所示:

```
storm nimbus >/dev/null 2>&1 &
```

在 Storm 集群的 Slave 节点上启动 Supervisor 进程, 执行命令如下所示:

```
storm supervisor >/dev/null 2>&1 &
```

在 Storm 集群的 Master 节点上启动 Storm 的 Web UI 服务进程, 执行命令如下所示:

```
storm ui >/dev/null 2>&1 &
```

你可以随时通过 jps 命令查看各个节点上的 Storm 服务进程是否正常启动, Nimbus 进程的显示名称为 nimbus, 同理 Supervisor 进程的显示名称为 supervisor, 而 Storm 的 Web UI 进程的显示名称为 core。然后请打开浏览器访问 <http://master:8080/index.html>, 你就能看到 Storm 的 Web UI 界面, 这里的 master 表示你的 Storm Master 节点所在服务器的 IP 或者域名。至此, Storm 集群就算搭建成功了。在开始 Solr 与 Storm 集成之前, 你需要了解如何编写与运行一个 Storm topology。

Storm 中的任务都需要包装成一个 Topology, 然后将 Topology 打包成 jar, 通过 storm 脚本上传提交给 Storm 的 Nimbus 服务进程, 最后由 Nimbus 服务进程分配到整个 Storm 集群的其他 Supervisor 进程去执行。

首先你需要在 Maven 的 pom.xml 中添加 storm-core 依赖, 配置示例如下:

```
<dependency>
<groupId>org.apache.storm</groupId>
```

```
<artifactId>storm-core</artifactId>
<version>${storm.version}</version>
<scope>provided</scope>
</dependency>
```

基于 Java 的 Storm Topology 由 3 个组件组成：

- ❑ Spouts: 它负责从外部数据源读取数据，然后发射数据流到 topology。
- ❑ Bolts: 处理 Spouts 或其他 Bolt 发射的数据流，然后发射出一个或多个数据流。
- ❑ Topology: 定义 Spouts 和 Bolts 如何安排，已经提供 topology 的入口点。

你需要分别实现自己的 Spout、Bolt、Topology，然后将其打包成 jar，然后通过 Storm 的脚本提交到 Storm 集群进行执行，提交的命令如下所示：

```
storm jar mywordcount.jar com.xx.xx.MyTopology arg1 arg2 arg3
```

其中 mywordcount.jar 是你自定义的 Storm Topology 打包成的 jar 包，com.xx.xx.MyTopology 是你的 Topology 实现类完整包的路径，其中 main 方法就是 Topology 的执行入口，arg1、arg2 和 arg3 为 com.xx.xx.MyTopology 类的 main 方法执行时可能需要传入的运行参数。关于如何自定义 Storm 的 Topology，随书源码中我提供了基于 Storm 实现 Word Count 统计的示例程序，仅供大家学习参考，Storm 方面的其他知识超出了本书范畴，这里不多讲了。

通过阅读随书源码中提供的 Word Count 示例程序，我相信你应该已经知道如何使用 Storm 向 Solr 中提交索引。你只需要在 Spout 中获取外部数据源（可能来自硬盘文件，可能来自关系型数据库），然后发射交给 Bolt，Bolt 再获取数据生成 SolrInputDocument，最后在 Topology 中获取每一个 SolrInputDocument 并通过 SolrClient 提交到 Solr Server 进行索引创建。具体实现就留给大家自己动手去实现了。

通过上面的思路确实可以实现将 Storm 从外部数据源接收的数据索引至 Solr 中，但是你可能需要考虑如下几个问题：

- ❑ 我们的实际业务代码如何与 Storm 代码实现分离，降低业务代码与 Storm API 的耦合度。
- ❑ 如何对 Bolt 和 Spout 的代码逻辑进行单元测试。
- ❑ 如何实现 Bolt 的参数可配置化，就好比 SolrCloud 需要的 Zookeeper 集群连接字符串。
- ❑ 如何打包并部署你的 Topology 至 Storm 集群。
- ❑ 如何在运行时对你的自定义的 Topology 进行性能测试。
- ❑ 如何与其他服务和数据库进行无缝集成。
- ❑ 如何在你的自定义 Topology 中将一个 Tuple 映射为 Solr 可以处理的数据格式。

当你使用 Storm 真正构建一个高性能的流式处理应用程序时，面临的问题可能更多，上面列举的问题仅仅只是冰山一角。因此，你可能希望有一个工具包能够简化 Storm 与 Solr 之间的集成，同时帮你解决上述的那些问题。由 Lucidworks 开源的 storm-solr 开源项目完美解决了这些问题，该项目的 Github 访问地址为 <https://github.com/lucidworks/storm-solr>。该项目目前是基于 Storm-0.9.4 + Solr-5.5.1 构建，不过我已经将其升级到 Storm-1.0.2 + Solr-

6.2.1. 如果你正在使用 Storm1.x 和 Solr6.x, 可以访问我的 Github 获取相关源码: <https://github.com/yida-lxw>。

想要在自己的项目中使用 storm-solr, 首先从 Github 上下载源码文件, 然后导入 IDEA 或 Eclipse, 对其编译并打包成一个 jar 包, 最后会自动在 target 目录下生成一个 original-storm-solr-2.2.2.jar, 你需要使用 mvn install 命令将其安装到本地 Maven 仓库再添加 POM 依赖, 同时 storm-solr 项目的 dependency 部分需要 copy 到你项目的 pom.xml 中, 当然可能会有部分依赖重复或者冲突, 请自己分析 Maven 依赖树然后进行部分排除。

storm-solr 项目提供了一个 com.lucidworks.storm.StreamingApp 类, 它允许你在本地或者远程 Storm 集群运行一个 Storm topology, 尤其是 StreamingApp 提供了如下几个功能:

- ❑ 分离了在不同环境中 Storm topology 的定义, 不同运行环境下的运行参数可以通过配置文件进行配置, 开发不同的 topology 只需要专注于实际业务需求, 帮你实现 Storm API 代码重复率最小化。
- ❑ 想要使用 StreamingApp 类, 你可以简单地实现 StormTopologyFactory 接口, 在其 build 方法中定义你的 spout、bolt, 最终返回一个 topology。
- ❑ 对于 Spout, 你需要自己实现 StreamingDataProvider 接口, 在其 open 方法中从外部数据源加载数据, 然后重写 next 方法判断当前数据是否已经发射完毕, storm-solr 内置实现的 SpringSpout 会循环调用 StreamingDataProvider 的 next 方法并不断向 Bolt 发射数据, 直到 next() 返回 false。
- ❑ 至于 Storm Bolt, 你不需要自己实现 Bolt, storm-solr 内置的 SpringBolt 能满足 Storm 与 Solr 数据交互的基本需求, 你只需要为 SpringBolt 设置不同的 SolrBoltAction 实现即可。
- ❑ 自定义 StormTopologyFactory 完成之后, 你可以调用 StreamingApp 的 StreamingApp (StormTopologyFactory topo, String[] args) 构造函数来初始化 StreamingApp 实例, 然后执行 StreamingApp 实例的 run 方法来运行 Storm topology。运行参数可以通过 -D 方式来传递, 当然你也可以在 classpath 下添加一个 Config.groovy 配置文件, 针对不同的开发环境配置不同的运行参数, 并将运行参数配置在配置文件中, 比直接使用 -D 方式更加便利, 默认支持 test (测试阶段)、development (开发阶段)、staging (演示阶段)、production (产品阶段), 具体请参阅 storm-solr 源码中提供的 Config.groovy 配置示例。
- ❑ storm-solr 中的一些主要实例对象全部使用 Spring IOC 来管理, 因此你还需要在 classpath 下添加一个 spring 的配置文件, 用来配置 storm-solr 的一些核心组件类, 比如 StreamDataProvider、StreamingDataAction、SolrInputDocumentMapper (用于处理 Tuple 与 SolrInputDocument 之间的映射)、DocumentAssignmentStrategy (用于处理指定的 SolrInputDocument 应该发送给哪个 Collection)。

除了可以直接运行 StreamingApp 类的 run 方法来完成 Storm topology 的运行测试。也可以在命令行下通过 storm-solr.jar 来运行 Storm topology, 执行命令示例如下所示:

```
java -classpath $STORM_HOME/lib/*:target/storm-solr-2.2.2.jar com.lucidworks.storm.StreamingApp com.xx.xxx.MyToSolrTopology -localRunSecs 90 -env staging
```


上面 `-classpath` 用于指定 `storm-solr.jar` 运行依赖的 `storm jar` 包, `MyToSolrTopology` 表示你自定义的 `StormTopologyFactory` 类的完整包路径, `-localRunSecs` 参数表示运行 90 秒后程序自动停止, `-env` 用于指定当前运行环境, `staging` 表示当前处于演示环境, 支持的其他可选参数值前面已经列举过。至此, 关于 Storm 如何添加索引数据至 Solr 中的方法, 就介绍到这里了。具体使用过程中, 建议大家多多阅读 `storm-solr` 的源码去理解经过 `storm-solr` 封装之后的 Storm Topology 是如何工作的。

16.14 Spark 与 Solr 进行数据交互

Spark 是 UC Berkeley AMP lab 开源的一款类似 Hadoop MapReduce 的通用并行分布式计算框架, Spark 基于 Map Reduce 算法实现分布式计算, 拥有 Hadoop MapReduce 所具有的优点; 但不同于 MapReduce 的是 Job 的中间输出和结果可以保存在内存中, 从而不再需要读写 HDFS, 因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 Map Reduce 算法。

Spark 提供的数据集操作类型有很多种, 不像 Hadoop 只提供了 Map 和 Reduce 两种操作。比如 `map`、`filter`、`flatMap`、`sample`、`groupByKey`、`reduceByKey`、`union`、`join`、`cogroup`、`mapValues`、`sort`、`partitionBy` 等多种操作类型, Spark 把这些操作称为 Transformations。同时还提供 `Count`、`collect`、`reduce`、`lookup`、`save` 等多种 actions 操作。

这些多种多样的数据集操作类型, 给开发上层应用的用户提供了方便。各个处理节点之间的通信模型不再像 Hadoop 那样就是唯一的 Data Shuffle 一种模式。用户可以命名, 物化, 控制中间结果的存储、分区等。可以说编程模型比 Hadoop 更灵活。同时 Spark 通过提供丰富的 Scala、Java、Python API 及交互式 Shell 来提高 Spark 的可用性。Spark 还可以直接对 HDFS 上的文件进行数据读写, 同样支持 Spark on YARN。Spark 可以与 MapReduce 运行于同集群中, 共享存储资源与计算, 数据仓库 Shark 实现上借用来 Hive, 几乎与 Hive 完全兼容。你可以将 Spark 内存计算之后的结果写入 HBase、Hive、Solr 等。

由于 Spark 源码是基于 Scala 编写的, 因此你需要先安装 Scala。Scala 安装包请自己从 Scala 官网下载, 这里建议下载 `tgz` 格式, Windows 环境下的童鞋请不要下载 `msi` 格式。跟安装 JDK 类似, 先解压安装包, 然后配置 Scala 环境变量。这里我以 Scala2.11.8 版本为例, 因为在我写本章节时, 当前 Spark 最新版本为 2.0.2, 且依赖于 Scala2.11.x。

```
# 配置 Scala 环境变量
SCALA_HOME=/opt/modules/scala-2.11.8 // 同时在 PATH 末尾追加:$SCALA_HOME/bin
scala -version // 使用此命令验证 Scala 是否安装成功
```

然后你需要安装 Spark, 这里为了简单起见, 我就搭建一个 Spark 伪分布式即 Master 与 Slaves 全部在同一台机器上。首先请从 Spark 官网下载安装包, 这里我以 Spark2.0.2 版本为例对 Spark 伪分布式集群安装过程进行讲解说明。以下是安装过程中涉及到具体操作:

```

sudo tar -zxf spark-2.0.2-bin-hadoop2.7.tgz -C /opt/modules/ // 解压安装包
sudo mv spark-2.0.2-bin-hadoop2.7/ spark-2.0.2-hadoop2.7/ // 将目录重命名, 可选的
sudo chown -R hadoop:hadoop spark-2.0.2-hadoop2.7/ // 设置目录的所有者为 hadoop
用户, 避免直接使用 root 用户操作
# 配置 Spark 环境变量
SPARK_HOME=/opt/modules/spark-2.0.2-hadoop2.7 // 在 PATH 环境变量的末尾追加:
$SPARK_HOME/bin:$SPARK_HOME/sbin

```

进入 /opt/modules/spark-2.0.2-hadoop2.7/conf 目录, 配置 spark-env.sh, 默认 spark-env.sh 文件并不存在, 将 spark-env.sh.template 文件重命名为 spark-env.sh, 然后再编辑配置。配置示例如下:

```

JAVA_HOME=/opt/modules/jdk1.8.0_111
SCALA_HOME=/opt/modules/scala-2.11.8
SPARK_MASTER_IP=linux.yida01.com // Master 节点的 ip 或域名
SPARK_MASTER_PORT=7077 // Master 节点监听的端口号
SPARK_MASTER_WEBUI_PORT=8080 // Master 节点的 Web UI 监听的端口号
SPARK_WORKER_CORES=2 // 当前节点上 Worker 实例的核数
SPARK_WORKER_MEMORY=2g // Worker 实例的内存大小
SPARK_WORKER_PORT=7078 // Worker 监听的端口号
SPARK_WORKER_WEBUI_PORT=8081 // Worker 的 Web UI 监听端口号
SPARK_WORKER_INSTANCES=1 // Worker 实例个数
HADOOP_CONF_DIR=/opt/modules/hadoop-2.7.0/etc/hadoop // Hadoop 的配置文件目录
SPARK_LOCAL_IP=linux.yida01.com // 当前 Spark 节点的 IP 或域名
SPARK_CONF_DIR=/opt/modules/spark-2.0.2-hadoop2.7/conf // Spark 的配置文件目录
SPARK_LOG_DIR=/opt/modules/spark-2.0.2-hadoop2.7/logs // Spark 的日志文件存放目录
SPARK_PID_DIR=/opt/modules/spark-2.0.2-hadoop2.7/pid // Spark 的进程文件存放目录
SPARK_IDENT_STRING=mySpark // Spark 的唯一标识符, 需要保证全局唯一, 默认值为当前系统用户
SPARK_NICENESS=0 // 设置守护进程的调度优先级 (默认为 0)
#SPARK_CLASSPATH= // 用于设置 Spark 依赖的其他 jar 包的加载路径

```

然后请不要忘记在 /opt/modules/spark-2.0.2-hadoop2.7 目录下创建 logs 和 pid 目录。特别注意, 不要忘记在 spark-env.sh 中配置 JAVA_HOME 和 SCALA_HOME。然后我们需要配置 Slave 节点, 由于这里我们配置的是伪分布式, 因此, 在 slaves 配置文件里只需填入当前机器的域名即可, 由于 slaves 文件并不直接存在, 你需要将 slaves.template 文件重命名为 slaves。然后先启动 HDFS 的 namemode 和 datamode, 然后再分别启动 Spark 的 Master 和 Slave。最后打开浏览器访问 <http://linux.yida01.com:8080/>, 查看 Spark 的 Web UI 界面。如果能够正常打开, 则表明 Spark 的伪分布式集群搭建成功了。

如果你需要在 Windows 机器上开发 Spark 应用程序, 那么还需要在 Windows 机器上安装 Scala, 与 Linux 上的安装步骤类似, Windows 上的配置环境变量稍微有点不同。你需要为你的 IntelliJ IDEA 安装 Scala 插件, 因为默认 IntelliJ IDEA 并不支持 Scala。由于我比较喜欢使用 IntelliJ IDEA, 因此如果有部分同学习惯使用 Eclipse, 那么请自己查找资料了解如何在 Eclipse 中安装 Scala 插件。以下是 IntelliJ IDEA 中安装 Scala 插件的详细步骤:

```
File --> Settings --> Plugins
```

输入 scala 关键字, Category 选择 Languages, 找到 Scala 的 IDEA 插件, 选中它, 然后单击右边绿色的 Install 按钮进行插件安装。安装过程可能会持续大概 10 分钟的样子, 请耐心等待, 安装完毕之后请重启你的 IntelliJ IDEA。

想要在 IntelliJ IDEA 中开发 Spark 应用程序, 首先你需要创建一个 Maven Project (当然普通的 Java Project 也可以)。然后你需要为你的 Project 添加 Scala 依赖。具体操作为: 请在 IntelliJ IDEA 左侧的 Project 视图中的项目名称上鼠标右键 → Add Framework Support, 然后在左侧勾选 Scala, 在右边单击 Create 按钮选择你的 Scala 本地安装路径。在项目的 pom.xml 上鼠标右键 → Maven → Reimport。最后在 pom.xml 中添加 hadoop-client 和 spark-core 依赖, 这里我使用的是 hadoop 版本为 2.7.0, 因为官网提供的 Spark2.0.2 版本是基于 Hadoop2.7.0 构建的。具体这两个依赖 pom.xml 中如何添加请查阅随书源码中的 pom.xml。

编写 Spark 应用程序, 首先需要构建一个 SparkContext 对象, 然后根据 SparkContext 对象可以加载本地文件或者 HDFS 上的文件来构造一个 RDD, 或者直接通过 makeRDD 函数来手动构建一个 RDD, 变换 RDD 将其每个元素映射成 SolrInputDocument 或者一个纯粹的普通类对象, 创建 CloudSolrClient 对象, 批量添加索引文档并 commit 到 Solr Server 来创建索引, 示例代码如下:

```
val sparkConf: SparkConf = new SparkConf()
    .setAppName("myScalaSolrApp")
    .setMaster("spark://linux.yida01.com:7077")
    .set("spark.default.parallelism", "1")
    .setJars(Array[String]("file:///E:/git-space/solr-book/target/scala-solr-test.jar"))

val sc: SparkContext = new SparkContext(sparkConf)
sc.addJar("libs/solr-solrj-6.2.1.jar")
sc.addJar("libs/noggit-0.6.jar")
sc.addJar("libs/httpmime-4.4.1.jar")
val rdd: RDD[String] = sc.textFile(hdfsPath, 2)
val rdd2: RDD[Array[String]] = rdd.map((i: String) => i.split(","))
val docList = rdd2.map(x => {
    val id = x(0)
    val firstname = x(1)
    val lastname = x(2)
    val age = x(3).toInt
    val doc = new SolrInputDocument();
    doc.addField("id", id);
    doc.addField("firstname", firstname);
    doc.addField("lastname", lastname);
    doc.addField("age", age);
    doc
})

val cloudSolrClient = new CloudSolrClient(zkHost)
cloudSolrClient.add(docList.toJavaRDD().toLocalIterator)
cloudSolrClient.setDefaultCollection(collection);
```

```
cloudSolrClient.commit(true, true);
cloudSolrClient.close();
```

不过，这里推荐你使用 Lucidworks 开源的 spark-solr 项目来实现 spark 与 solr 的数据交互，通过使用 spark-solr 能够简化 Spark 与 Solr 之间的数据交换，比如 Spark 中将 RDD 数据索引至 Solr，或者读取 Solr 索引数据返回 RDD，之后针对 RDD 做各种内存计算。此外，spark-solr 天然支持对普通的 TXT、JSON 或 CSV 格式数据进行索引，支持直接对关系型数据库中的数据进行索引，而且代码极其简洁。

你首先需要下载 spark-solr 源码，下载地址：<https://github.com/lucidworks/spark-solr>，然后执行 maven 命令进行源码编译打包，执行命令如下：

```
install -X -Dmaven.test.skip=true
```

当然你也可以直接通过 Maven 仓库引入 spark-solr 依赖，比如：

```
<dependency>
<groupId>com.lucidworks.spark</groupId>
<artifactId>spark-solr</artifactId>
<version>3.0.0-alpha.1</version>
</dependency>
```

不过，在我写当前章节时，spark-solr-3.0.0-alpha.1 支持的 Spark 版本是 2.0.1，如果想要支持更高版本的 Spark，那么你就需要导入源码自己编译。一般为了便于学习，建议你导入源码到 IDEA 或者 Eclipse 中，因为源码中提供了很多示例程序，只有编译通过了，才能运行这些示例。

首先创建一个 SparkSession 对象，示例代码如下：

```
var sparkSession = SparkSession.builder()
    .appName("spark-solr-test")
    .setMaster("spark://linux.yida01.com:7077")
    .config("spark.default.parallelism", "1")
    .getOrCreate()
```

当然你也可以通过 `var sc = sparkSession.sparkContext` 来获取原生的 SparkContext 对象。通过 SparkSession 对象你可以直接读取文件并转换成 RDD，比如加载 CSV 文件：

```
val csvDF = sparkSession.read.format("com.databricks.spark.csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .load(csvFileLocation)
```

如果需要直接读取 JSON 文件，你可以这样：

```
val moviesDF = sparkSession.read.json("/xx/xxxxx.json")
```

你还可以直接加载 Solr 中的某些 Collection 的数据，比如这样：

```
val solrOpts = Map("zkhost" -> zkHost, "collection" -> s"$collection1Name,$collection2Name")
val solrDF = sparkSession.read.format("solr").options(solrOpts).load()
```

获取到 `DataFrame` 对象之后，你可以将其提交到 Solr 进行索引，示例如下：

```
csvDF.write.format("solr").options(solrOpts).mode(Overwrite).save()
val solrCloudClient = SolrSupport.getCachedCloudClient(zkHost)
solrCloudClient.commit(collectionName, true, true)
```

如果你的数据是 Java 数组或者 `java.util.List` 类型，你可以这样提交数据至 Solr 并创建索引，示例代码如下所示：

```
String[] inputDocs = new String[] {
    "1,foo,bar",
    "2,foo,baz",
    "3,bar,baz"
};
// 将一个普通的 Java 数组转换成 JavaRDD 对象
JavaRDD<String> input = jssc.sparkContext().parallelize(Arrays.
asList(inputDocs), 1);
LinkedBlockingDeque<JavaRDD<String>> queue = new LinkedBlockingDeque<JavaRDD<S
tring>>();
queue.add(input);
// JavaRDD 对象转成 JavaDStream 对象
JavaDStream<SolrInputDocument> docs = jssc.queueStream(queue).map(
    new Function<String, SolrInputDocument>() {
        public SolrInputDocument call(String row) {
            // 将每个 String 映射成一个 SolrInputDocument 对象
            String[] fields = row.split(",");
            SolrInputDocument doc = new SolrInputDocument();
            doc.setField("id", fields[0]);
            doc.setField("field1", fields[1]);
            doc.setField("field2", fields[2]);
            return doc;
        }
    });
// 提交 JavaDStream 对象到 Solr Server 并索引
SolrSupport.indexDStreamOfDocs(zkHost, collectionName, 1, docs.dstream());
```

如果想要将一个 Java 对象或者一个 Java 对象集合提交到 Solr Server 并创建索引，那么此时你需要 Java 对象转换成一个 `RDD[SolrInputDocument]` 对象，同理 Java 对象集合需要转换成 `Iterable[SolrInputDocument]` 对象。

如果你想要执行 Solr 查询，也很简单，只需要通过 `SparkSession` 对象进行 `load` 操作即可，示例代码如下：

```
val searchExpr = s""
|search(${collectionName},
```

```

      |      q="the_abcdefghijklmnopqrstuvwxyz_field_01:[* TO *]",
      |      fl="{f1}",
      |      sort="the_abcdefghijklmnopqrstuvwxyz_field_01 asc",
      |      qt="/select")
var myDataFrame = sparkSession.read.format("solr").options(
  Map("zkhost" -> zkHost, "collection" -> ${collectionName}, "expr" -> searchExpr)).
load

```

你甚至可以使用 SQL 语句来执行 Solr 查询，示例如下：

```

val sqlStmt =
  s"""
    | SELECT movie_id, COUNT(*) as agg_count, avg(rating) as avg_rating, sum
(rating) as sum_rating, min(rating) as min_rating, max(rating) as max_rating,
    | FROM ${ratingsCollection}
    | GROUP BY movie_id
    | ORDER BY movie_id asc
  """
  .stripMargin
var sqlDF = sparkSession.read.format("solr").options(
  Map("zkhost" -> zkHost, "sql" -> sqlStmt)).load

```

执行 Solr 查询过后，你会得到一个 DataFrame 对象，DataFrame 对象提供了很多工具方法来完成进一步的结果集处理，比如过滤、分组、排序等，常用的几个操作如下所示：

```

selectExpr      // 执行 solr 表达式查询
filter          // 执行结果集过滤
groupBy         // 执行分组操作
sort            // 执行排序操作
count           // 统计结果集总数
select          // 返回指定的域
collect.foreach // 遍历结果集

```

经过这些函数处理之后的 DataFrame 对象还可以再次写入到 Solr 索引中，示例如下：

```

myDataFrame.write
  .format("solr")
  .options(Map("zkhost" -> zkHost, "collection" -> ${collectionName}, "batch_
size" -> "10000"))
  .save

```

此外 spark-solr 还支持添加 Shard、Shard 分割等操作，暂时还不支持其他操作，比如删除索引、实时 Get 查询。关于 spark-solr 的开源项目就介绍到这里，其实主要是 Spark 环境搭建，使用起来还是很简单的，当然这需要你有一定的 Scala 语言基础以及 Spark 框架的基本了解。在搭建 Spark 开发环境时，尤其需要注意的是，你 Windows 开发机器上使用的 Spark 和 Scala 版本务必与你服务器上安装的 Spark 和 Scala 版本保持一致，否则运行时可能会抛异常。我给出的这些提示权当抛砖引玉，我也不可能细致入微的介绍 spark-solr 的方方面面，实际还是需要大家自己去动手导入 spark-solr 源码，细细品读从而理解其底层是如何封装的，从而能够帮助你更好的使用它。然后多多动手编写示例代码去测试去实践。我也

希望通过本章的内容能够激发你对 Spark、Scala 的学习热情，附带的能够给你一些提示，让你在学习路上少走一些弯路就已足矣。

16.15 Solr6 中的 SQL 接口

在 Solr6.x 中针对 SolrCloud 模式提供了 Parallel SQL Interface (并行 SQL 接口) 新特性。此 SQL 接口无缝的结合了 SQL 与 Solr 的全文检索功能。对于 aggregations (聚合操作) 有两种实现：类似 MapReduce 的 Shuffle 和 JSON Facet API。使用哪种方式取决于你的性能需求。这些特性使得 Solr 的 SQL 接口可以广泛应用于各种使用场景。

16.15.1 Solr SQL 架构

Solr SQL 接口允许你发送一个 SQL 查询到 Solr Server，然后你将在响应结果中得到返回的流式 Document。Solr SQL 接口是基于 Presto Project 的 SQL Parser (SQL 语法解析器) 构建，该 SQL 语法解析器能够实时将 SQL 查询翻译成 Streaming 表达式。

在标准的 SELECT SQL 语句中比如 "SELECT <expressions> FROM <table>", 表名相当于 Solr 的 Collection 名称，表名是不区分大小写的，而 SQL 语句中的列名直接被映射为被查询的 Collection 的索引中的域名称，列名是区分大小写的，同时支持 SQL 中的别名，并且别名可以在 ORDER BY 子句中引用。不支持使用 SQL 中的通配符 * 语法返回所有 Field，可以在 SQL 语句中返回 Solr 中的 score 这个伪域，但是前提是 SQL 中必须包含 LIMIT 子句。例如，我们可以索引一些 Solr Document，然后构造类似如下的 SQL 查询：

```
SELECT manu as mfr, price as retail FROM techproducts
```

这里的 techproducts 即 Solr 中的某个 Collection 名称，然后请求返回 manu 和 price 这两个域，同时使用 as 关键字以别名形式返回。

Solr 中的 SQL 功能可以采用如下两种方式来执行 aggregation (聚合) 操作：

□ map_reduce：这种实现会通过 shuffle 操作将 tuple 分发给 Yarn 平台上的 Worker 节点去执行。其中涉及到对整个结果集的排序和分区操作，然后分发给各个 Worker 节点。采用这种方式时，tuple 被发送到 Worker 节点之后会按照 GROUP BY 域进行排序，然后 Worker 节点汇总合并某个组的 aggregate 结果。这种方式使得用户可以无限制的执行 aggregation 操作，但是你不能跨多个 Worker 节点在网络间传输整个结果集。

□ facet：这种方式是使用 JSON Facet API 或者 StatsComponent 组件来实现 aggregation 操作。在这种情况下，aggregations 操作逻辑会被推送到查询引擎，仅仅只是 aggregate 部分需要跨网络传输，这是 Solr 操作的常规模式。当 GROUP BY 的域的基数很小时，这种方式执行速度会很快。如果你的 GROUP BY 域中包含了大基

数域（域的唯一域值个数很多就叫大基数域，比如 boolean 类型域的域值就 true 和 false 两种，那么它就是小基数域），那么可能会导致程序崩溃。这些模式你可以在给 Solr 发送请求时通过 aggregationMode 属性进行传递。选择哪种 aggregation 模式这取决于你 GROUP BY 域的基数，如果你 GROUP BY 域的基数很小，那么此时选择 "facet" 模式性能会更高。反之你就需要使用 "map_reduce" 模式。

Solr SQL 架构主要分 3 层进行设计，其中 SQL 层处于 /sql 请求处理器端，它负责接收 Solr SQL 查询请求，并且将其转换成并行查询计划。选择 N 个 Worker 接口来执行此查询计划，然后将查询计划发送给这 N 个 Worker 节点去并行执行。一旦查询计划在 Worker 节点上被执行，/sql 请求处理器会对这 N 个 Worker 节点返回的 Tuple 执行最终的合并操作。

Worker 层的 Worker 节点的主要职责是接收 /sql 请求处理器发送过来的查询计划然后并行执行它。并行执行计划中包含的查询比如需要在 Data Table（数据表）层执行的 SQL 以及关系代数计算。每个 Worker 节点被分配的查询任务是从 Data Table 中的 Tuple 随机 Shuffle 1/N。

Data Table（数据表）层处于 Table 端，每个 Table 对应着 SolrCloud 中的一个 Collection。Data Table 层负责接收来自 Worker 节点的查询以及 Tuple，同时还需要处理 Tuple 的初始化排序和分区，然后将 Tuple 发送给 Worker 节点。这意味着 Tuple 在进行网络传输之前是已经被排序并分区的，然后将每个分区的 Tuple 直接按序发送给正确的 Worker 节点，随之进入 Reduce 准备阶段。

下面是对图 16-3 的执行流程详细描述：

- ❑ /sql 请求处理器首先接收 Client 的 SQL 查询请求，并解析 SQL 查询同时创建并行查询计划。
- ❑ /sql 请求处理器将查询计划发送给 Worker 节点（图中绿色部分）。
- ❑ Worker 节点并行执行查询计划，图中显示每个 Worker 节点会连接 Data Table 层的某个 Collection。Data Table 层的 Collection 就是 SQL 查询的 Table。注意，图中的 Collection 拥有 5 个 Shard 并且每个 Shard 拥有 3 个 Replica。每个 Worker 节点连接每个 Shard 中的一个 Replica。因为图中有 5 个 Shard，因此每个 Worker 节点会返回来自每个 Shard 的结果集，占整个查询结果集的 1/5。数据分区是在 Data Table 层完成，因此不存在出现跨网络数据重复。同时还要注意，Data Table 层的所有 Replica 都是事先同时经过排序和分区的，
- ❑ Worker 节点并行处理来自 Data Table 层的 Tuple，Worker 节点可能还需要执行关系代数计算来满足查询计划。
- ❑ Worker 节点处理完毕后会将 Tuple 以流式返回给 /sql 请求处理器。
- ❑ /sql 请求处理器对每个 Worker 节点返回的 Tuple 执行合并操作。
- ❑ 最后 /sql 请求处理器将合并后的 Tuple 以流式返回给客户端。

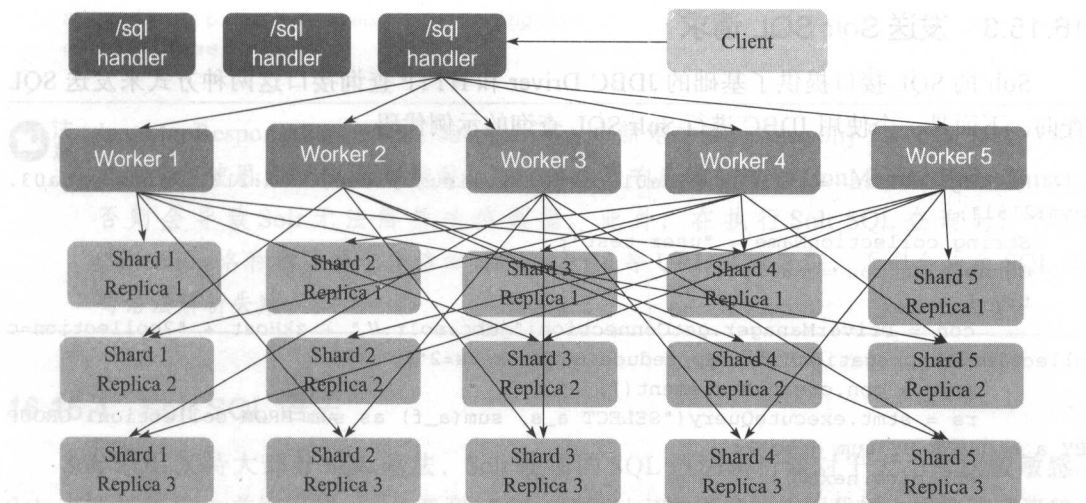



图 16-3 Solr SQL 架构图

16.15.2 Solr SQL 配置

Solr SQL 接口使用的 Request Handler 默认已经被隐式配置，这意味着 Solr SQL 功能你不需要做过多前置准备工作并且它一直处于被启用状态。

Solr SQL 接口的请求处理器是 `/sql`。所有的 SQL 查询请求都将被 `/sql` 请求处理器所接收并处理。在 `map_reduce` 模式下运行 `GROUP BY` 和 `SELECT DISTINCT` 查询时，此请求处理器同时会协调 MapReduce Job 的分布。默认 `/sql` 请求处理器会选择拥有 Collection 的 worker 节点来处理分布式操作。在这种默认情况下，`/sql` 请求处理器端驻留的 Collection 将扮演着 MapReduce 查询的默认 Worker Collection。

 **注意** 如果你拥有高基数的域并且拥有大量数据，这里建议你为 SQL 查询单独建立一个 Collection。

Solr 中的 Streaming API 是一个用于 SolrCloud 的可扩展的并行计算框架。Streaming 表达式提供了一种查询语言以及一种用于 Streaming API 的序列化格式。Streaming API 支持在大数据集上快速的 MapReduce 来并行处理关系代数计算。Solr SQL 接口会首先使用 Presto SQL Parser 来解析 SQL 查询，然后将其转化成并行查询计划。并行查询计划使用 Streaming API 与 Streaming 表达式来表示。正如 /sql 请求处理器一样，Solr 同样会隐式配置 /stream 请求处理器。

在某些场景下，SQL 查询中使用的 Field 必须配置为 DocValue 域，如果查询没有 limit 子句，那么所有域都必须是 DocValue 域，如果查询中包含 limit 子句，那么 SQL 查询中使用的 Field 不必开启 DocValues。

16.15.3 发送 Solr SQL 请求

Solr 的 SQL 接口提供了基础的 JDBC Driver 和 HTTP 查询接口这两种方式来发送 SQL 查询, 下面是一个使用 JDBC 进行 Solr SQL 查询的示例代码:

```
String zkHost = "linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181";
String collectionName = "user-test";
Connection con = null;
try {
    con = DriverManager.getConnection("jdbc:solr://" + zkHost + "?collection=collection1&aggregationMode=map_reduce&numWorkers=2");
    stmt = con.createStatement();
    rs = stmt.executeQuery("SELECT a_s, sum(a_f) as sum FROM collection1 GROUP BY a_s ORDER BY sum desc");
    while(rs.next()) {
        String a_s = rs.getString("a_s");
        double s = rs.getDouble("sum");
    }
} finally {
    rs.close();
    stmt.close();
    con.close();
}
```


注意, 要使上面的示例代码正确执行, 你必须在 pom.xml 中添加如下依赖且版本号必须是 6.x:

```
<dependency>
<groupId>org.apache.solr</groupId>
<artifactId>solr-solrj</artifactId>
<version>6.2.1</version>
</dependency>
```

Solr Server 通过 /sql 请求处理器来接收并行 SQL 查询请求, 下面是一个通过 SolrJ 发送 Solr SQL 查询请求的示例代码:

```
private static final String ZK_HOST = "linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181";
private static final String COLLECTION_NAME = "logs";
CloudSolrClient client = createCloudSolrClient(ZK_HOST, COLLECTION_NAME);
SolrQuery solrQuery = new SolrQuery();
solrQuery.setRequestHandler("/sql");
solrQuery.set("aggregationMode", "facet");
solrQuery.set("collection", COLLECTION_NAME);
solrQuery.set("stmt", "SELECT id,log_level,log_msg FROM " + COLLECTION_NAME + " ORDER BY id desc LIMIT 10");
client.setParser(new DelegationTokenResponse.JsonMapResponseParser());
QueryResponse resp = client.query(solrQuery, SolrRequest.METHOD.POST);
NamedList<Object> nameList = resp.getResponse();
```


```
System.out.println(nameList.toString());
client.close();
```

 **注意** `JsonMapResponseParser` 类是 `solr-solrj-6.x` 中新增的类, `solr-solrj-5.x` 中没有这个类, 并且当你使用 Solr SQL 查询时, 你必须设置响应解析器为 `JsonMapResponseParser`, 否则会导致 Solr 无法解析响应数据。此外, 在执行 Solr SQL 查询时, 你的 `Collection` 名称请不要包含诸如 `-&%?#@*!d` 等 URL 特殊字符, 否则会导致 SQL 语句语法解析失败。

16.15.4 Solr SQL 语法

Solr SQL 支持大部分 SQL 语法, Solr 使用的 SQL 语法解析器对于大小写比较敏感。Solr 支持包含 `limit` 关键字的 `select` 查询, SQL 中的 `limit` 关键字的作用我想大家应该都清楚。然而在 Solr SQL 中加不加 `limit` 关键字会导致生成不同的查询执行计划以及对 Solr 索引中的域的属性设置也有不同的要求, 比如 SQL 语句中不包含 `limit` 关键字的话, 那么 Solr 索引中的域的 `docValues` 属性必须设置为 `true`。下面是一个 Solr SQL 语句简单示例:

```
SELECT fieldA as fa, fieldB as fb, fieldC as fc FROM tableA WHERE fieldC =
'term1 term2' ORDER BY fa desc LIMIT 100
```

 **注意** Solr SQL 中的 `WHERE` 子句中必须在一侧包含域, 同时不能两侧同时包含域, 因此 `WHERE 5 < 10` 和 `WHERE fieldA > fieldB` 都是不支持的, 同时 SQL 子查询在 Solr SQL 中暂时也不支持。

`WHERE` 语句支持如下几种写法:

```
WHERE fieldC = 'term1 term2'
WHERE fieldC = '(term1 term2)'
WHERE fieldC = '[0 TO 100]'
WHERE fieldD>10
WHERE ((fieldC = 'term1' AND fieldA = 'term2') OR (fieldB = 'term3'))
WHERE (fieldA = 'term1') AND NOT (fieldB = 'term2')
```

不难发现, 其实与 Solr 中的简单查询表达式语法相同。注意, `WHERE fieldC in(...)` 暂时不支持这种 `in` 关键字查询, 但是你可以使用 `WHERE fieldC=(term1 term2 term3 term4)` 这种迂回的方式实现。Solr SQL 支持标准的 SQL 语法中的操作符如表 16-3 所示。

表 16-3 Solr SQL 支持的 SQL 语法操作符

操作符	示 例	Solr 等价查询	描 述
=	fielda = 10	fielda:10	等值查询

(续)

操作符	示 例	Solr 等价查询	描 述
<>	fieldA <> 10	-fieldA:10	不等于查询
!=	fieldA != 10	-fieldA:10	不等于查询
>	fieldA > 10	fieldA:{10 TO *}	大于查询
>=	fieldA >= 10	fieldA:[10 TO *]	大于等于查询
<	fieldA < 10	fieldA:[* TO 10}	小于查询
<=	fieldA <= 10	fieldA:[* TO 10]	小于等于查询

Solr SQL 中暂时不支持 BETWEEN、LIKE、IN 等 SQL 关键字，然而，你可以采用变通的方式来实现 BETWEEN 和 LIKE。对于 BETWEEN，你可以采用 Range Query 来实现，比如 price = [50 TO 100]。对于 LIKE，你可以采用通配符查询来实现，比如 username='tom*'。

ORDER BY 子句支持按多个域排序，同时还支持 score 伪域，但是前提是 SQL 语句中包含了 limit 关键字。ORDER BY 的域名称是区分大小写的。

Solr SQL 还支持 SQL 中的 SELECT DISTINCT 查询，示例如下：

```
SELECT distinct fieldA as fa, fieldB as fb FROM tableA ORDER BY fa desc, fb desc
```

Solr SQL 接口还支持针对数字域进行简单的统计计算，支持的 SQL 函数有 count(*)、min、max、sum、avg 等。因为这些函数不需要对数据进行 Shuffle 操作，这些聚合操作最终会被 Solr 的 StatsComponent 组件来负责处理。Solr SQL 中使用 SQL 聚合函数的简单示例如下所示：

```
SELECT count(fieldA) as count, sum(fieldB) as sum FROM tableA WHERE fieldC = 'Hello'
```

Solr SQL 中还支持 GROUP BY、HAVING 等查询子句，示例如下所示：

```
SELECT fieldA as fa, fieldB as fb, count(*) as count, sum(fieldC) as sum,
avg(fieldY) as avg FROM tableA WHERE fieldC = 'term1 term2'
GROUP BY fa, fb HAVING sum > 1000 ORDER BY sum asc LIMIT 100
#HAVING 和 ORDER BY 子句中还可以包含聚合函数
SELECT fieldA, fieldB, count(*), sum(fieldC), avg(fieldY) FROM tableA
WHERE fieldC = 'term1 term2' GROUP BY fieldA, fieldB
HAVING ((sum(fieldC) > 1000) AND (avg(fieldY) <= 10))
ORDER BY sum(fieldC) asc LIMIT 100
```

16.15.5 Solr SQL 客户端可视化工具的使用

Solr SQL 接口支持接收来自 SQL Client、数据库可视化工具（例如 DbVisualizer、Apache Zeppelin）的查询请求。Solr SQL 客户端连接 Solr Server 的连接字符串与 MySQL 的 JDBC 连接字符串很相似，示例如下：

```
jdbc:solr:// SOLR_ZK_CONNECTION_STRING?collection=COLLECTION_NAME
```


其中 aggregationMode 和 numWorkers 连接参数是可选的, aggregationMode 参数用于配置聚合操作的工作模式, 可选值有 map-reduce 和 facet。numWorkers 参数用于配置 Worker 节点的个数。

Solr SQL 支持的客户端工具有 DbVisualizer、Squirrel SQL、Apache Zeppelin (还处于孵化期)。这里以 DbVisualizer 工具为例讲解如何使用它连接 Solr Server, 就好比 we 使用 Navicat for MySQL 工具连接 MySQL 数据库。首先从 DbVisualizer 的官网 (<https://www.dbvis.com/>) 下载安装包, 然后进行安装。安装完成之后, 请单击顶部菜单栏 Tool 中的 Driver Manager 项, 打开 Driver 配置界面, 单击大致左上角的绿色加号按钮创建一个新的 Driver。在右侧填写 name、URL Format。name 表示 Driver 名称, 这个可以随意定义。URL Format 表示 Solr SQL 客户端的连接字符串的格式, 请填写为 jdbc:solr://<zk_connection_string>/?collection=<collection>。然后你需要添加 Solr SQL Driver 依赖的一些 jar 包, 如图 16-4 所示, 依赖的 jar 包有: \$SOLR_HOME/dist/solrj-lib 目录下的所有 jar 包以及 \$SOLR_HOME/dist/ 目录下的 solr-solrj-<version>.jar。

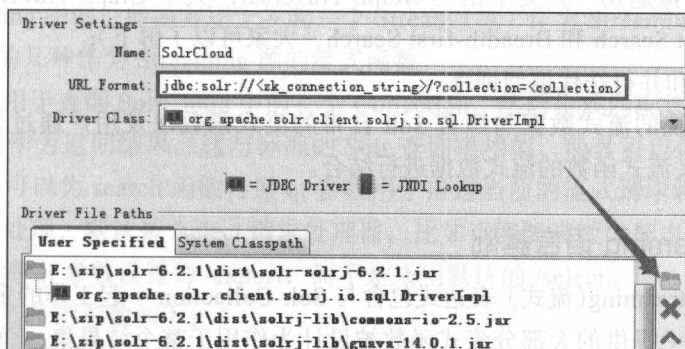


图 16-4 添加 Solr SQL Driver 依赖 jar 包示意图

如图 16-5 所示, 创建一个 Connection, 按照创建向导一步步去操作。第一步要求你填写 Connection 名称, 这个随意填写。第二步要求你选择一个 DataBase Driver, 这里选择刚刚创建的 Driver。第三步请填写 DataBase URL, 填写示例如下:

```
jdbc:solr:// linux.yida01.com:2181,linux.yida02.com:2181,linux.yida03.com:2181?collection=canal
```

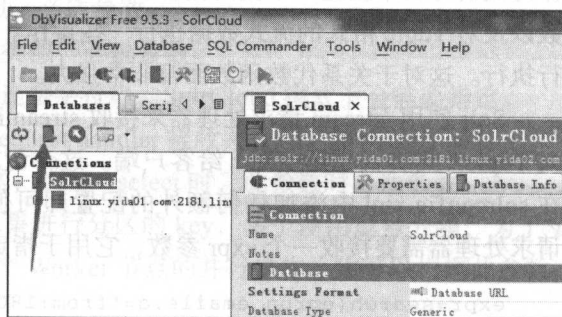


图 16-5 创建一个 Solr SQL 的 JDBC 连接示意图

连接创建成功之后, 就可以单击上方的 SQL Commander 菜单选择其中的 New SQL Commander。然后就可以在打开的界面中编写 SQL 语句, 单击第一个绿色的三角状按钮执

行 SQL 语句来查询 Solr Server 中的索引数据。

16.16 Solr6 中的 Streaming 表达式

Solr6 中的 Streaming 表达式提供了一种用于 SolrCloud 模式下的简单而强大的流式处理语言。Solr 中提供了一套函数集，它们可以结合在一起执行各种不同的并行计算任务，并且这些函数也是 Solr SQL 并行接口的基础。这些可用的函数如下所示：

- 请求与响应的流式处理；
- 批量流式处理；
- 快速的交互式 MapReduce；
- Aggregation（聚合操作）；
- 并行的关系代数计算（分布式的 join、intersections、unions、complement）；
- 消息发布与订阅；
- 分布式图像遍历（英文术语为 Graph Traversal，关于 Graph Traversal 有两种算法：Depth-first Search 和 Breadth-first Search，大家可以了解下）；
- 机器学习和并行迭代模型训练。

来自外部系统的流式数据与源自 Solr 自带的流式函数以及用户通过 Solr 的 Streaming API 添加的自定义流式函数的流式数据进行结合。

16.16.1 Streaming 语言基础

Solr6 中的 Streaming（流式）表达式包含与 Solr Collection 一起工作的多种 Streaming（流式）函数。Solr6 中提供的大部分流式函数被设计为作用于整个结果集，而不仅仅是普通查询返回的 Top N 结果集。Solr 中的 /export 请求处理器支持这种流式操作。一些流式函数作为流的源头来发起一个流的流程。而其他的流式函数则作为流的装饰者来包装其他流式函数以及对 Tuple 格式的流式数据执行一些操作。大部分流式函数支持跨 Worker Collection 并行执行。这对于关系代数函数而言非常强大。

Solr 使用 /stream 请求处理器来接收 streaming 表达式请求并且返回一个 JSON 格式的流式 Tuple（其实就是键值对）给客户端。这个请求处理器是隐式定义的，这意味着你不需要在 solrconfig.xml 中添加任何额外的配置即可使用 Solr6 中的 Streaming 功能特性。/stream 请求处理器需要接收一个 expr 参数，它用于指定 streaming 表达式，示例如下：

```
expr=search(enron_emails,q="from:1800flowers*",fl="from, to", sort="from asc",qt="/export")
http://localhost:8983/solr/${collectionName}/stream
```

在 SolrJ6.x 的 org.apache.solr.client.solrj.io 包下提供了一些类用于将 streaming 表达式编译成 streaming 接口对象，使用示例如下所示：

```
StreamFactory streamFactory = new StreamFactory().withCollectionZkHost("collection1", zkServer.getZkAddress())
    .withStreamFunction("search", CloudSolrStream.class)
    .withStreamFunction("unique", UniqueStream.class)
    .withStreamFunction("top", RankStream.class)
    .withStreamFunction("group", ReducerStream.class)
    .withStreamFunction("parallel", ParallelStream.class);
ParallelStream pstream = (ParallelStream)streamFactory.constructStream("parallel(collection1, group(search(collection1, q=\"*:*\", fl=\"id,a_s,a_i,a_f\", sort=\"a_s asc,a_f asc\", partitionKeys=\"a_s\"), by=\"a_s asc\"), workers=\"2\", zkHost=\""+zkHost+"\", sort=\"a_s asc\")");
```

由于 Solr6 中的 Streaming 表达式依赖于 /export 请求处理器，而 /export 请求处理器对域定义有一定的要求，比如域的 docValues 属性必须为 true 等，详细内容请翻阅本书的 13.6 节进行回顾了解。

16.16.2 Streaming 源函数

Solr Streaming 中的一些函数用于发起一个 Stream（流）作为 Streaming 源，下面依次介绍 Solr6 中常用的几种作为 Streaming 源的流式函数。

search 函数用于查询 SolrCloud 中的一个 Collection，然后生成匹配该查询的一个 Tuple 形式的流式数据作为返回结果。这与标准的 Solr 查询很相似，两者可以使用大部分相同的查询参数。你还可以为 search 函数传递 qt 参数用于指定当前的流式请求将由 Solr 的哪个请求处理器接收并处理，默认是 /select 请求处理器。比如当需要高效的导出整个 Collection 的数据时，你需要将 qt 参数设置为 /export，而不是采用默认的 /select。

search 函数支持的参数如下：

- ❑ collection：需要查询的目标 Collection 名称，必选参数。
- ❑ q：主查询表达式，与 Solr 标准查询里的 q 参数类似，必选参数。
- ❑ fl：与 Solr 标准查询里的 fl 参数类似，必选参数。
- ❑ sort：与 Solr 标准查询里的 sort 参数类似，必选参数。
- ❑ zkHost：用于指定 Zookeeper 集群的连接字符串，SolrCloud 模式下会需要指定。
- ❑ qt：用于设置当前流式请求由哪个 Request handler 接收并处理，默认值为 /select。
- ❑ rows：用于设置返回返回前 N 条数据，当 qt=/select 时，此参数时必选参数。
- ❑ partitionKeys：用于指定对查询结果集进行分区的 key，多个使用逗号分隔，为了实现同 parallel 函数一起使用实现跨多个 Worker 节点的并行操作，你可以指定此参数。

search 函数语法示例如下所示：

```
expr=search(collection1,zkHost="localhost:9983",qt="/export",
    q="*:*",fl="id,a_s,a_i,a_f",sort="a_f asc, a_i asc")
```

jdbc 函数用于查询 JDBC 数据源并生成一个 Tuple 形式的流式数据来表示 JDBC 查询结

果集。结果集中的每一行会被转换成一个 Tuple，而每个 Tuple 包含每行的每一列的数据。Jdbc 函数支持的参数如下所示：

- ❑ connection：JDBC 格式的连接字符串，比如 jdbc:mysql://localhost/users?user=root&password=solr，必选参数。
- ❑ sql：从 JDBC 数据源加载数据需要指定的 SQL 语句，必选参数。
- ❑ sort：指定排序条件，比如 price desc，必选参数。
- ❑ driver：指定你的 JDBC 连接使用的 JDBC 驱动类，比如 com.mysql.jdbc.Driver。
- ❑ driverProperty：用于设置传递给 JDBC 驱动的一些属性，设置格式为 propertyName="propertyValue"。比如 characterEncoding="utf8",useUnicode="true",connectTimeout="5000"。

下面是一个 jdbc 函数的简单使用示例：

```
jdbc(
  connection="jdbc:mysql://localhost/users?user=root&password=solr",
  sql="SELECT id, name FROM users",
  sort="id asc",
  driver="com.mysql.jdbc.Driver"
)
```

facet 函数用于提供了以流式来处理 Facet 聚合操作，其实在 Solr Server 端还是借助于 Solr 的 JSON Facet API 来完成 Facet 操作。facet 函数极其适合于对小基数域进行聚合操作，如果需要对大基数域进行聚合操作，那么你需要使用 rollup 函数。facet 函数支持的参数如下所示：

- ❑ collection：你需要对哪个 Collection 执行 facet 聚合操作，必选参数。
- ❑ q：用于构建 facet 聚合操作的 Solr 查询表达式，必选参数。
- ❑ buckets：逗号分隔的域，多个域采用逗号分隔，表示在多个维度上进行 Facet 聚合操作。
- ❑ bucketSorts：逗号分隔的多个排序规则，排序依据可以是 buckets 参数中某个域的值，也可以是某个函数的计算值。
- ❑ bucketSizeLimit：用于限制 buckets 的数量，这个参数应用于每个维度。
- ❑ metrics：应用于每个 Bucket 的函数计算，当前支持 sum(col), avg(col), min(col), max(col), count(*)。

facet 函数的一个简单使用示例如下所示：

```
facet(collection1,
  q="*:* ",
  buckets="year_i, month_i, day_i",
  bucketSorts="year_i desc, month_i desc, day_i desc",
  bucketSizeLimit=100,
  sum(a_i),min(a_i),max(a_i),avg(a_i),count(*) )
```

topic 函数提供了构建于 SolrCloud 之上的发布 / 订阅消息功能。topic 函数允许用户订

阅某个查询，然后此函数会向用户投递匹配该 Topic 查询的最新更新的 Document。初次调用 topic 函数会为指定的 TopicID 创建检查点，后续调用涉及同一个 Topic ID 会返回自该初始检查点之后新增或更新的 Document。每次执行 Topic 查询都会更新该 Topic ID 对应的检查点。设置 initialCheckpoint 参数为 0 会导致该 Topic 会处理匹配该 Topic 查询的所有 Document。Topic 函数支持的参数如下所示：

- ❑ checkpointCollection: topic 检查点存储在哪个 Collection，必选参数。
- ❑ collection: 在哪个 Collection 上执行 Topic 查询，必选参数。
- ❑ id: Topic 的唯一标识符，每个检查点对应一个 Topic ID。必选参数。
- ❑ q: Topic 查询表达式，必选参数。
- ❑ fl: Topic 查询返回的域列表。必选参数。
- ❑ initialCheckpoint: 用于设置初始检查点即初始的 _version_ 版本号值，topic 函数会从该起始版本号开始从队列读取索引，如果没有设置那么默认会从当前最高版本开始。此参数设置为 0 则会处理 Topic 查询 (q 参数表示的查询) 匹配到的所有索引文档。

一个 topic 函数的简单使用示例，如下所示：

```
topic(checkpointCollection,
      collection,id="uniqueId",q="topic query",fl="id, name, country")
```

16.16.3 Streaming 装饰函数

Solr Streaming 中还有一些函数用于装饰另一个流式函数，并在该流式函数生成的流式数据基础上执行一些操作，比如更新索引、执行两个流式函数生成的流式数据的合并操作、排序操作、两个流式函数生成的流式数据的 Join 操作（类似标准 SQL 里多表连接操作）。

update 函数用于装饰另一个流式函数并将该函数生成的流式数据发送至 Solr Server 来进行索引创建。以下是一个 update 函数的简单使用示例：

```
update(destinationCollection,
      batchSize=500,
      search(collection1,
        q="*:*", fl="id,a_s,a_i,a_f,s_multi,i_multi", sort="a_f asc, a_i asc"))
```

以上的函数示例表示将 search 函数执行后返回的流式数据以 500 个索引文档为一个批次批量添加到 destinationCollection 这个 Collection 中进行索引创建。

```
daemon(id="uniqueId",
      runInterval="1000",terminate="true",
      update(destinationCollection,batchSize=100,
        topic(checkpointCollection,
          topicCollection,q="topic query",
            fl="id, title, abstract, text",
```

```
id="topicId",initialCheckpoint=0)
    )
}
```

以上的示例表示会启动一个守护进程，每间隔 1000 毫秒调用一次 update 函数，而 update 函数订阅了指定 topicId 的 Topic，用于接收新增或更新的 document，然后进行索引，即增量索引更新操作。整体来说就是每间隔 1 秒执行一次增量更新，uniqueId 用于唯一标识一个 daemon 操作实例，后续可以使用此 ID 来 kill 此 daemon 函数操作线程。Terminate 参数表示当 Topic 不再发送 Tuple 时是否应该终止当前守护线程。当然你也可以配置为 jdbc 函数启动守护线程周期性调用来实现增量索引关系型数据库表里的数据，示例如下所示：

```
daemon(id="12345", runInterval="60000",
    update(users, batchSize=10,
        jdbc(connection="jdbc:mysql://localhost/users?user=root&password=solr",
            sql="SELECT id, name FROM users", sort="id asc", driver="com.mysql.jdbc.
Driver")
    ))
```

Solr 为此还提供了一套 API 接口用于控制 Daemon 操作，比如：

```
# 查看当前正在运行的 daemon 流式函数操作
http://localhost:8983/solr/users/stream?action=list
# 根据 daemon ID 来启动、停止指定的 Daemon 函数操作
http://localhost:8983/solr/users/stream?action=start&id=123456
http://localhost:8983/solr/users/stream?action=stop&id=12345
# 根据 daemon ID 从 Solr 中彻底 kill 掉 Daemon 流式操作实例
http://localhost:8983/solr/users/stream?action=kill&id=123456
```

executor 函数用于包装另一个流式函数并开辟多个线程以并行的方式执行该函数。比如你可以使用 executor 函数来包装 update 函数实现多线程更新索引至 Solr Server。executor 函数开辟的多个线程是运行于同一个 Worker 节点上的，如果你需要跨多个 Worker 节点并行执行，那么可以使用 parallel 函数来包装 executor 函数。可以为 executor 函数指定 threads 参数来明确指示开辟多少个线程。executor 函数使用示例：executor(threads=10,update(.....))。

```
top(n=3,search(.....))
```

top 函数用于从其他流式函数匹配的结果集中提取 Top N 个结果，类似 SQL 里的 limit 关键字。

```
select(
    search(collection1, fl="id,teamName_s,wins,losses", q="*:*" , sort="id asc",
        teamName_s as teamName, wins,
        replace(wins,null,withValue=0)))
```

select 函数用于重新修改另一个函数返回的域，比如为其设置别名，或者替换域原来的值再返回等，比如上面示例中的 replace 函数会判断 wins 域是否为 null，如果为 null 就替换为 0。


```

commit(
    destinationCollection,
    batchSize=2,           // 批量提交的索引文档个数
    waitFlush:false,       // 是否一直阻塞直到索引真正写入到硬盘, 默认值 false
    waitSearcher:false,    // 是否一直阻塞直到 IndexSearcher 实例初始化完毕, 默认值 false
    softCommit:true,       // 是否开启软提交, 默认值 false
    update(
        destinationCollection, // 提交到哪个目标 Collection
        batchSize=5,
        search(...)))

```

commit 函数用于提交一个索引更新操作:

```

fetch(category,
    search(book, q="*:*", fl="id,bookName,categoryId", sort="id desc"),
    fl="id, categoryName", on="categoryId=id")

```

这个 fetch 函数操作会遍历 search 函数匹配的结果集, 然后根据 book.categoryId 与 category.id 进行关联来提取 category 里的所有数据:

```

complement(
    search(...),
    search(...),,
    on="fieldA=fieldB"
)

```

complement 函数用于求函数 A 和函数 B 的补集 (即返回 A 中不包含在 B 中的元素)。on 参数用于指定某个域来判断两者是否相等, 相等则表明 A 中该元素存在于 B。如果两者比较的域名称相同, 则 on 参数可以简写为 on="fieldA"。同理还有 intersect 函数用于求 A、B 的交集。

```

innerJoin(
    search(people, q="*:*", fl="personId,name", sort="personId asc"),
    search(pets, q="type:cat", fl="ownerId,petName", sort="ownerId asc"),
    on="personId=ownerId"
)

```

innerJoin 函数执行的操作就好比 SQL 里的 innerJoin, 例如 select * from A innerjoin B on A.aID = B.bID。同理还有 leftOuterJoin 函数, 类似 SQL 里的左外连接。

```

sort(innerJoin(
    search(people, q="*:*", fl="id,name", sort="id asc"),
    search(pets, q="type:dog", fl="owner,petName", sort="owner asc"),
    on="id=owner"
), by="name asc, petName asc")

```

sort 函数用于对另一个流式函数匹配的结果集进行排序, 排序规则通过 by 参数进行指定。关于 Solr6 中的 Streaming 流式函数就介绍这么多, 其他更多函数请大家访问 Solr 的官方

Wiki 作进一步补充了解：<https://cwiki.apache.org/confluence/display/solr/Streaming+Expressions>。当然你也使用 SolrJ 来执行 Solr6 中的流式函数请求操作，具体示例请大家查阅随书源码的第 16 章。

16.17 Solr 常见问题解答

在使用 Solr 的过程中，你一定遇到过各种各样的问题。不论部署环境是简单还是复杂，不论是运行于单机还是 SolrCloud 模式，你都或多或少会碰到一些棘手的问题。在本章节中，我将针对一些大家普遍比较关心的问题进行解答。

Q1：如何正确设置 SOLR_HOME

Solr 中，SOLR_HOME 路径决定了你的 Core 或 Collection 实际存放的位置。在单机模式下，你创建的所有 Core 都必须在 SOLR_HOME 目录下；而在 SolrCloud 模式下，你的 Collection 的所有 Replica 副本将会存放在各个节点的 SOLR_HOME 目录下。设置 SOLR_HOME 变量就好比安装 JDK 需要设置 JAVA_HOME。正确设置 SOLR_HOME 能够指示 Solr 将所有数据存放至你指定的目录，这样更便于你日后管理维护。

在 Solr 中配置 SOLR_HOME 有如下几种方式：

在 Tomcat 的 server.xml 中的 <Host> 元素之间添加如下配置：

```
<Context docBase="webapps/solr.war" debug="0" crossContext="true">
  <Environment name="solr/home" type="java.lang.String" value="/opt/solr_home"
  override="true" />
</Context>
```

或者在 solr 的 WEB-INF\web.xml 配置文件中的 <web-app> 元素之间添加如下配置：

```
<env-entry>
  <env-entry-name>solr/home</env-entry-name>
  <env-entry-value>/usr/solr</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

或者通过设置 JVM 启动参数方式设置，比如 -Dsolr.solr.home=/opt/solr_home。如果使用 Tomcat 部署 Solr，那么你可以在 Tomcat 的 catalina.sh 配置文件中添加如下配置：

```
JAVA_OPTS="$JAVA_OPTS -Dsolr.solr.home=/opt/solr_home"
```

如使用 Jetty 部署 Solr，那么你可以在 solr.in.sh 配置文件中配置 solr.solr.home 启动参数，配置示例如下：

```
SOLR_OPTS="$SOLR_OPTS -Dsolr.solr.home=/opt/solr_home -Dsolr.clustering.
enabled=true-Dbootstrap_conf=true"
```

当然也可以在启动 Solr 时临时指定 -Dsolr.solr.home 这个启动参数，比如当 solr 脚本启

动 Solr 时，你可以指定 `-s` 参数，具体请翻阅本书的 14.4.4 节，详细了解 Solr 提供的 solr 脚本使用参数。

Q2: Solr 的时区问题

Solr 的时区问题已经困扰大家很久，对于中国人而言，十分不习惯索引的时间数据实际存储格式 (UTC)，与我们相差 8 小时，每次查阅数据脑子里还得瞬间转换成北京时间，这简直不能忍。为此我翻阅 Solr 源码，发现 Solr 中的日期处理全部是由 `org.apache.solr.util` 包下的 `DateMathParser` 类进行处理，而该类内部默认采用的却是 UTC。在 Solr 中日期时间类型，我们一般使用 `TrieDateField` 类型，而 `TrieDateField` 在处理日期时间字符串转换成 `java.util.Date` 对象时，严格限制了日期时间格式字符串必须是 UTC 格式，即必须包含字符“Z”，这部分源码在 `DateMathParser` 类的 `parseMath (Date now, String val)` 方法中。因此，首先我们需要更改 `DateMathParser` 类的默认时区并重写 `parseMath` 方法，突破默认日期时间字符串的 UTC 格式限制。然后 Solr 查询返回的结果集中的索引文档如果也包含了 `TrieDateField` 域，此时对于 JSON 格式而言，数据解析工作是由 `JSONResponseWriter` 类处理，而 `JSONResponseWriter` 类的 `writeDate` 方法中会通过 JDK8 中的 `Instant` 类的 `toString` 方法来完成 `Date` 到 `String` 的转换工作，而 `Instant` 类的 `toString` 方法中有如下这样一段代码：

```
public String toString() {
    return DateTimeFormatter.ISO_INSTANT.format(this);
}
```

你会发现实际采用的是 ISO8601 时间格式标准来格式化 `Date` 对象，但是这是 JDK8 中的内部行为，我们只能重写 `JSONResponseWriter` 类的 `writeDate` 方法，摒弃 JDK8 中的 `Instant` 实现。

以上就是我解决 Solr 时区问题的思路，具体实现源码请访问我 Github 上的 `solr-timezone` 项目，访问地址如下：

<https://github.com/yida-lxw/solr-timezone>

你只要将 `solr-timezone` 项目源码导入 IDEA 或 Eclipse 中，对其编译打包成一个 jar 包，然后将其导入 solr 的 `core` 或者 `collection` 中，当然 `solr-timezone` 项目自身还需要依赖 solrJ 和 jackson 相关 jar 包，也需要一并导入。之后在 `schema.xml` 中使用自定义的 `TrieDateField` 域类型，而不是 Solr 内置的 `TrieDateField`，如果你需要 Solr 查询结果集中返回的日期格式也能够以国人习惯的日期时间格式返回，那么你还需要在 `solrconfig.xml` 中配置自定义的 `JSONResponseWriter`。具体配置示例请查阅 Github 上的 README 文档。

最后还需要设置 JDK 的 `user.timezone` 属性，这里有 2 种方式进行设置。如果使用的是 Jetty，你可以在 `solr.in.sh` 中配置 `SOLR_TIMEZONE="Asia/Shanghai"`，或者在 `SOLR_OPTS` 参数中进行配置，示例如下：

```
SOLR_OPTS="$SOLR_OPTS -Dsoler.clustering.enabled=true -Dbootstrap_conf=true
```

```
-Duser.timezone=Asia/Shanghai"
```

如果使用的是 Tomcat，那么你需要在 Tomcat 的 `catalina.sh` 中添加如下配置：

```
TOMCAT_TIMEZONE="-Duser.timezone=Asia/Shanghai"
export CATALINA_OPTS="$CATALINA_OPTS $TOMCAT_TIMEZONE"
```

设置完成之后，请重新加载你的 Core，如果是在 SolrCloud 模式下，你需要在每个节点上进行修改，然后依次重启每个 Solr 节点。最后可以通过 Solr 的 Web UI 界面左侧的 Java Properties 菜单提供的 Java 相关的系统属性，请查找“`user.timezone`”这项并检查其参数值是否为 `Asia/Shanghai`。

Q3: Solr 拼音分词问题

有时候你希望用户输入拼音就能搜索到结果，那么你需要对查询文本进行拼音分词。这里说的拼音分词指的是将汉字转成拼音并对拼音进行索引。当然也可以直接对汉语拼音进行分词索引，比如将“`jianchidaodi`”分成“`jianchi daodi`”。对汉字进行拼音分词一般的做法是采用 Lucene 的同义词原理来实现。大致就是将拼音 Token 的位置增量设置为 0，即将 `PositionIncrement` 设置为 0。如果有拼音分词这方面的需求，你可以使用我扩展后的 IK 分词器，Github 访问地址为：

<https://github.com/yida-lxw/IK>

天然支持拼音分词的分词器还有 HanLP、Jcseg，具体请翻阅本书的 4.3 节进行详细了解。但是需要注意的是，如果想要为某个域使用拼音分词，那么你需要为该域设置 `termPositions="true"`、`termOffsets="true"`。

Q4: Solr 如何禁用对索引文档进行打分操作

如果只需要根据关键字进行搜索，不需要额外的根据文档评分计算的相关度对文档进行排序，那么此时你可以选择禁用对文档进行评分。具体解决方案就是自定义一个 `Similarity` 实现，打分函数统统 `return 1`。示例如下：

```
public class CustomSimilarity extends DefaultSimilarity {
    @Override
    public float tf(float freq) {
        return 1.0;
    }
    @Override
    public float tf(int freq) {
        return 1.0;
    }
    @Override
    public float idf(int docFreq, int numDocs) {
        return 1.0;
    }
}
```

然后 schema.xml 中配置 `<similarity class="myorg.mypackage.CustomSimilarity"/>` 如果你只想对某个域类型设置, 那么你需要在 `<fieldType>` 元素内 `</analyzer>` 之后设置 `<similarity>`。

Q5: Solr 字典文件太大, 无法上传到 Zookeeper 的问题

当你初次创建 Collection, Solr 会自动将 Collection 相关的配置文件上传至 Zookeeper, 来实现 Collection 的各个 Shard 的 Replica 副本能够共享一份配置文件。你可以将 schema.xml、solrconfig.xml、data-config.xml、停用词字典等文件上传至 Zookeeper。但是需要注意, Zookeeper 默认限制单个文件不能超过 1MB, 否则会提示你: `IOException: Unreasonable length=xxx`。因为 Zookeeper 设计初衷就是为了实现文件存储的。Zookeeper 只适合存储一些小的数据块。比如一些程序中的配置参数、配置文件等。虽然你可以通过在 zoo.cfg 配置文件中通过调整 `jute.maxbuffer=10M` 参数来增大上传文件的大小。请注意, 每个节点都需要修改并且修改完成之后请记得重启各个 Zookeeper 节点。但是为了 Zookeeper 运行性能考虑, 请不要上传过大的配置文件至 Zookeeper。比如用户自定义字典文件可能有几十 M, 但是你又想要将其在各个 Zookeeper 节点之间共享, 此时你可以将其切分成多个小文件再上传共享。或者你可以使用 Solr 提供的 API 接口来共享同义词或停用词字典文件, 具体请翻阅 4.2.3 节 StopFilter、SynonymFilter 部分的内容。

Q6: Solr 字典文件如何动态更新

由于 Solr 中的配置文件、字典文件是在 Solr Core 或 Collection 初始化时被加载至内存中的, 后续都是直接访问内存而不是重新读写磁盘上的文件。因为你对磁盘上对配置文件或字典文件进行修改并不会立即对当前 Core 或 Collection 可见。因此想要对配置文件或字典文件的修改能够立即可见, 你必须在修改完毕之后重新加载当前 Core 或 Collection, 而这是 Solr 配置文件或字典文件动态更新的关键。比如你的配置文件可能是由程序动态去修改的, 如果你想要修改能够立即可见, 那么你可以周期性的加载配置文件然后重新加载 Core 或 Collection。但是 Core 或 Collection 重新加载过程中依然是按照旧的配置来处理客户端请求。Solr 提供了 HTTP 接口来实现 Core 或 Collection 的管理, 关于 Core 的 HTTP 接口请查阅 2.1.3 章节, 关于 Collection 的 HTTP 接口请查阅 14.8 章节。

Q7: 如何解决 Solr 索引文件被损坏的问题

假设公司有一个 Solr 集群由你搭建并维护, 突然有一天半夜你被告知 Solr 集群挂了, 提示 "the index is corrupted", 此时你需要及时分析并解决这个问题。想象下在半夜起来处理这种优先级很高的棘手问题是一件多么令人沮丧的事情。那么什么情况下会导致 Solr 索引被损坏呢, 主要有以下几种情形:

- ❑ 索引创建任务突然意外中断, 从而导致将部分构建的索引复制到生产环境中。
- ❑ 在索引复制期间网络连接突然中断, 从而导致一个或多个索引数据段文件损坏。
- ❑ 在建立索引或复制期间系统的文件描述符达到上限或磁盘空间不足, 从而导致索引损坏。

首先你需要尝试使用 `lucene-core-version.jar` 中的 `CheckIndex` 类来修复索引，你可以导入 `lucene-core` 这个 jar 包，然后创建 `CheckIndex` 对象并调用其 `doMain(String args[])` 方法，同时传入 `-fix` 运行参数来修复索引；或者直接在命令行输入如下命令来执行 `lucene-core.jar`：

```
java -cp lucene-core-${version}.jar org.apache.lucene.index.CheckIndex $INDEX_DIR -fix
```

上面的 `$INDEX_DIR` 参数表示你的索引目录。如果提示无法修复，那么你需要使用索引备份来替换之前的旧索引。关于 `Collection` 的备份与恢复请翻阅 14.8.13 节。由于 `Replica` 节点会自动与 `Shard Leader` 进行索引同步，因此一般你只需要恢复 `Leader` 节点即可。如果你连索引备份都没有，那只能尝试重新全量创建索引，比如你的索引数据是从外部数据库读取然后写入到 Solr 中，那么你需要再次重复这个步骤。这里强烈建议每天做好 `Collection` 的备份，以防万一。

Q8：如何解决 Solr 获取索引锁文件超时问题

根据前面学习的知识，我们可以知道：Solr 在创建索引的过程中，索引目录中的当前索引文件会被 Solr 锁住。假如由于一些不可预知的意外导致 Solr 索引创建过程中断，但是该文件锁会仍然被占用，从而导致你后续无法修改索引数据。比如你正在执行一个 `commit` 操作，突然你的 JVM 崩溃了，由于我们的 Solr 实例是运行于 JVM 之上，因此我们的 Solr 实例也会被销毁，然后我们的 Web 容器（比如 Jetty）会抛出如下异常：

```
SEVERE: Exception during commit/optimize:java.io.IOException: Lock obtain
timed out: SimpleFSLock@/opt/solr_home/solr/data/index/luceneff1fe872c2cbfeb4
4091b36c21a97c14-write.lock
```

如果你确定此刻没有其他线程正在往此索引目录创建索引，那么你可以切入到该索引目录下，找到一个文件名称以 `.lock` 结尾的文件，然后删除它，最后重启你的 Web 容器，问题即可得以解决。

如果当前有其他线程正在往此索引目录写索引，请暂时暂停该索引写入操作，然后删除所有目录下的 lock 锁文件，然后重启 Web 容器，最后再恢复索引创建线程即可。或者你可以临时改变该线程创建索引的索引存储目录，待问题修复之后再将索引合并到一起。

Q9：如何解决 too many open files 问题

当索引数据量很大，索引更新又很频繁时，你在创建索引过程中可能会遇到下面这样一个异常：

```
java.io.FileNotFoundException: /opt/solr_home/solr/data/index/_8.tii
This shows that there are too many open files.
```

首先修改 `/etc/security/limits.conf` 配置文件，调大系统文件描述符最大限制。虽然调整这个最大限制能够起到缓解作用。但是导致这个问题的罪魁祸首是索引创建过程中生成了大量的索引段文件。因此你需要定期的执行索引合并和优化工作来减小索引段文件个数，关于

这部分优化请翻阅 15.3 和 15.4 章节。

Q10: 如何解决 Solr 运行时抛出的 OutOfMemory 异常

OutOfMemoryException 是 Java 程序员司空见惯的异常，也是令人头痛的问题。一般都是由于 JVM 堆内存不足或者 GC 配置不合理导致的。而 Solr 自身又是一个极度依赖内存来提升查询性能的应用，因此你首先为 Solr 运行实例所处所在的 JVM 配置足够的堆内存。至于如何为 JVM 配置堆内存请翻阅 15.7 章节进行了解。然后你可以尝试使用 CMS 垃圾回收器或者 G1 垃圾回收器，同时开启多线程垃圾回收提高 GC 效率，可以适当调大 Survivor（幸存区）的所占比例大小，防止对象过早进入 Tenured 区域（即老年区）。但还是需要反复核查是否真的是 GC 导致的 OOM，为了方便查找真实原因，你需要开启 GC LOG，可以通过在启动命令中添加 `-Xloggc:gc.log` 参数，或者在 `solr.in.sh` 配置文件中添加。此配置参数表示会将 GC 的垃圾回收详细日志信息写入到该日志文件中，方便我们日后进行问题分析。配置 `-Xloggc` 与配置 `-Xmx-Xms` 参数类似，具体如何配置请翻阅 14.3.2 章节。

Q11: SolrCloud 模式下重启某个节点时间过长的的问题

当 Solr 集群中某个节点由于各种原因意外中断退出了，待问题修复之后我们需要重启该节点使其重新加入集群。但是你或许会遇到节点重启时间过长的的问题。这种问题大致可能由以下几个因素导致：

- ❑ 你的 Solr 事务日志大小：Replica 节点在重启时需要与 Shard Leader 进行数据同步。如果 Replica 节点离线时间很久，而且刚好在 Replica 节点离线的期间内，索引事务日志文件变更很大，那么 Replica 节点需要回放的索引操作很多，从而 Replica 节点重启需要消耗的时间会变得更长。加快自动提交频率可以减小你的事务日志体积大小，从而使得 Replica 节点数据恢复过程变得更快，但是提交间隔太短会影响你的索引性能。
- ❑ Replica 节点离线间隔时间：在 Replica 节点离线期间，默认当超过 100 条索引更新，此时甚至会导致 Replica 节点进行全量数据恢复，从而使得启动过程会消耗更多的时间。
- ❑ Overseer 队列请求阻塞：当你的 Replica 节点重启时，你的 overseer 队列中可能已经堆积了大量待处理的请求。你可以查看你的 Zookeeper 上的 `/overseer` 节点的队列大小。访问地址 `http://linux.yida01.com:8983/solr/#/~cloud?view=tree`，单击左侧的 Cloud 下的 tree 菜单，然后展开右侧的 `/overseer`，展开 "Metadata"，你会看到 `children_count` 这项信息。
- ❑ 可能存在的 Overseer 角色切换：在启动 Replica 节点时，通常建议一次只启动一个 Replica 节点。待一个 Replica 节点启动完成之后在进行下一个。同时你最好确保 Overseer 作为最后一个启动节点，毕竟切换 Overseer 角色也需要消耗时间。

默认索引更新超过 100 条会导致 Replica 节点进行全量数据恢复，自 Solr5.1 版本开始，

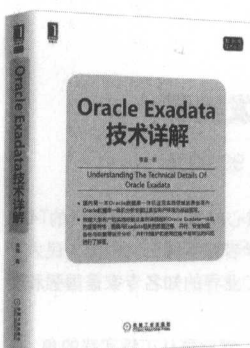
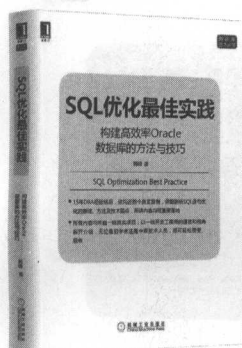
这个默认值 100 可以在 solrconfig.xml 中通过配置进行修改，配置示例如下所示：

```
<updateLog>  
  <str name="dir">${solr.ulog.dir}</str>  
  <str name="numRecordsToKeep">${solr.ulog.numRecordsToKeep:200}</str>  
</updateLog>
```

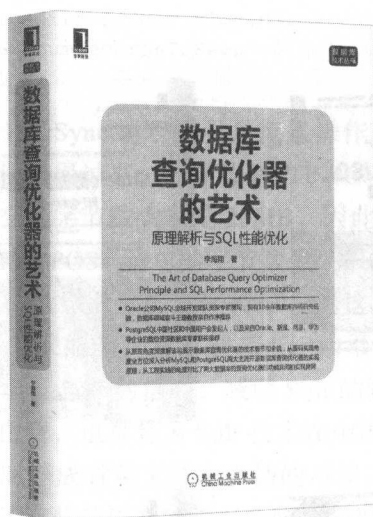
Solr 中的 PeerSync 请求并不是流式操作，它们是普通的 HTTP POST 请求，因此如果你将此参数值设置得过高，会导致 HTTP POST 请求失败。对于 Jetty 而言，HTTP POST 请求默认能够提交的字节数大小是 20MB。因此你在设置 numRecordsToKeep 参数值时需要确保不会超过 HTTP POST 请求的字节数最大限制。

很开心也很欣慰大家能够坚持学习到这里。同时本书也进入收尾阶段，我也很激动。希望通过本书大家能够真正掌握 Solr，如果能够在学习 Solr 方面激起大家学习 Solr 的兴趣，增添大家学习 Solr 的信心，减缓 Solr 的学习曲线，并切入实际地帮助到大家，那么我的付出就没有白费，也算是为 Solr 技术在中国地区的推广尽一点我个人的绵薄之力。这里，我建议大家平时养成看英文技术文档的习惯，比如官方提供的用户指南或者纯英文技术书籍。虽然这对于有些英语基础不好的同学会很痛苦，但是如果你能够坚持 1 年、2 年，你的学习能力将得到巨大的提升，而学习能力是你一生的财富。这里我强烈向大家推荐一款手机 APP：Safari Queue。该 APP 上拥有大量与 IT 相关的英文技术书籍，你只需要使用一个邮箱账号注册用户，然后就可以免费在线阅读书籍。唯一遗憾的是默认只有大概 30 天的免费期限，过期后你可以再重新注册一个账号。最后声明一点，我没有为该手机 APP 打广告，他们也没有给我广告费，是真心推荐，一般人我不告诉他。OK，就到这里了，祝大家学习愉快！

推荐阅读



推荐阅读



数据库查询优化器的艺术：原理解析与SQL性能优化

作者：李海翔 ISBN：978-7-111-44746-7 定价：89.00元

本书是数据库查询优化领域的里程碑之作，数据库领域泰斗王珊教授亲自作序推荐，PostgreSQL中国社区和中国用户会发起人以及来自Oracle、新浪、网易、华为等企业的数位资深数据库专家联袂推荐。

本书从原理角度深度解读和展示数据库查询优化器的技术细节和全貌；从源码实现角度全方位深入分析MySQL和PostgreSQL两大主流开源数据库查询优化器的实现原理；从工程实践的角度对比了两大数据库的查询优化器的功能异同和实现异同。它是所有数据开发工程师、内核工程师、DBA以及其他数据库相关工作人员值得反复研读的一本书。

数据库事务处理的艺术：事务管理与并发控制

作者：李海翔 ISBN：978-7-111-58235-9 定价：99.00元

作者有近20年数据库内核研发经验，曾是Oracle公司MySQL全球开发组核心成员，现在是腾讯的T4级专家。数据库领域的泰斗杜小勇老师亲自为本书作序，数据库学术界的知名学者张孝博士（中国人民大学）、卢卫博士后（中国人民大学）、彭煜玮博士（武汉大学），以及数据库工业界的知名专家盖国强和姜承尧等也给予了极高的评价。

全书共12章，首先介绍数据库事务管理与并发控制的基础理论和工作机制，然后再从工程实践的角度对比和分析了4个主流数据库的事务管理与并发控制的实现原理，最后通过源代码分析了PostgreSQL和MySQL在事务管理与并发控制上的技术架构。

作者简介

兰小伟（网名：益达） 资深Java工程师，在Java技术上有很深的积累和造诣。国内较早接触Solr的技术专家之一，长期致力于Solr的技术研究、实践和生产环境部署，是Solr社区的积极参与者和实践者，以让Solr技术能够在中国得到广泛应用不遗余力并乐此不疲。

现就职于国美金融，曾就职于各种大大小小的创业型公司。个人技术涉猎广泛，除了Java之外，对jQuery、ExtJS、AngularJS等前端技术也有研究。

技术宅，外表高冷安静，内心细腻感性，好文墨喜交友但不善交际。为人低调谦和，乐于助人，愿与各位志同道合者一同交流学习。

Solr是一个构建在Apache Lucene上的流行的、快速的、开源的企业搜索平台，它的主要功能包括强大的全文搜索、命中高亮、多维度查询与分析统计、丰富的文档解析、地理空间搜索、大量的REST API以及并行SQL。Solr是安全的、高度可伸缩的、可自动容错的分布式索引和搜索的企业级解决方案，并为世界上许多高流量的internet站点提供了全文搜索和导航的技术支持并备受欢迎。

Solr在企业内一个典型的应用场景就是电商商品搜索、类别导航区块、属性过滤区块、搜索框自动联想，等等。当下已是大数据时代，企业的业务数据量呈TB级增长，对于数据的搜索需求会愈加强烈，对于低成本的互联网企业，Solr的使用诉求也会更加普遍。

对于历史数据的查询，在数据量还不具规模的情况下，一般采用传统关系型数据库自带的索引功能即可实现高效的数据查询。但当数据上升到一定规模时，或许你会想到使用HBase数据库来救急，然而HBase目前只支持针对rowkey的一级索引，尚且不支持二级索引，此时Solr+Hbase的珠联璧合就可以完美打破这一局限性。但Solr的强大不仅如此，尤其当你足够了解Solr之后。

本书采用浅显易懂的语言加以适当的配图为你详细解读Solr的每个技术点，让其中涉及的每个原理、机制都不再晦涩难懂。理论结合实践才能出真知，案例驱动的方式贯穿本书始终，希望读者能够多上机实践书中的每个示例，遵循“理解为主，实践为辅”的学习原则，学以致用并在自己所在公司企业内部部署Solr，充分施展Solr的威力，从而体现自己的个人价值。



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机\程序设计

ISBN 978-7-111-58207-6



9 787111 582076 >

定价: 89.00元